BASIC09

BASIC09 Reference

Contents

Chapter 1

Looking at the Basics

BASIC09 is a computer language created for use with the OS-9 operating system. Along with standard BASIC language statements and functions, it includes the most useful elements of the PASCAL computer language.

In brief, BASIC09's advantages are:

Fast execution speed BASIC09 compiles procedure lines as you enter them. When you finish a procedure, you can compile it further. The result? Procedures that execute nearly as fast as machine language.

Full feature editing The text editor features automatic line formatting, search, search and change, global search, global search and change, line renumbering, and much more. You can move in and out of the editor quickly and easily.

Modular programming functions You can write small, easy-to-understand procedures, then chain them to create sophisticated programs. You can call one procedure from another, regardless of whether the called procedure is in memory or on disk.

Interfacing to OS-9 Both you and your procedures can take advantage of almost any OS-9 function from within BASIC, including the execution of disk management commands and application programs.

Structured programming You can structure procedures more efficiently and clearly by taking advantage of a variety of loop commands, optional line numbering, and BASIC09's ability to call modules written in other computer languages.

Memory saving features

Strings can be any length. For each operation, you can select the most efficient of five available data types. Compiled procedures use less space. You can save several procedures into one file.

Complex data structures

Combine any type of data into a single dimensioned data structure that you can move, store, and assign easily and quickly.

Sophisticated graphics

BASIC09 has three levels of graphics. The high resolution graphics and text capabilities feature more than 50 functions.

High speed, precision math

BASIC09 has a full range of fast and accurate math and transcendental capabilities including powers, roots, trigonometry, logic, and Boolean functions.

Simple and fast debugging

BASIC09 provides superior debugging functions. It checks syntax as you enter lines. It points to the location of your errors and tells you what they are. You can stop programs, enter the debugger, then continue execution. Execution errors automatically put you in a debugging mode where you can examine values, and step and trace your way through faulty procedures.

Using BASIC09

Before anything else, make a backup copy of your BASIC09/CONFIG diskette. You can do this using the BACKUP command. If you are not familiar with BACKUP, see Chapter 3 of *Getting Started With OS-9*.

To use BASIC09, boot your computer as described in *Getting Started With OS-9*. Replace the system diskette in Drive /D0 with the BASIC09/CONFIG backup diskette and type:

```
basic09 [ENTER]
```

After a short pause, during which OS-9 loads BASIC09 from the diskette, the screen displays the copyright and a new *prompt*, like this:

```
                     BASIC09
                RS VERSION 01.00.01
         COPYRIGHT 1980 BY MOTOROLA INC.
           AND MICROWARE SYSTEMS CORP.
            REPRODUCED UNDER LICENSE
                 TO TANDY CORP.
               ALL RIGHTS RESERVED

Basic09
Ready
B:
```

The B: indicates that your computer is in the BASIC09 *command mode*. From the command mode, you can issue instructions to the system executive to manipulate procedures (programs).

Requesting More Memory

Unless you specify otherwise, BASIC09 automatically sets aside 8192 bytes of memory as a workspace into which you can type or load procedures. BASIC09 reserves approximately 1200 bytes of the workspace for internal use, leaving you with 6992 bytes for workspace.

There are two ways to set aside more memory for BASIC09 operations:

- You can reserve extra memory when you first enter BASIC09 by using the OS-9 *memory size* option. For instance, to reserve 18,176 bytes, enter this command to initialize BASIC09:

  `basic09 #18k` [ENTER]

- You can also request additional memory after loading BASIC09. At the command prompt, `B:`, type:

  `mem 18000` [ENTER]

  This tells BASIC09 to set aside a total of 18,000 bytes of memory, if they are available.

In both cases, because BASIC09 rounds the amount you request to the next multiple of 256, the actual reserved memory is 18176 bytes.

> **Note:** If your system does not have enough free memory to reserve the amount you specify, the workspace size does not change.

You can also use the MEM command to reduce memory. However, BASIC09 does not reduce the size of the workspace if doing so destroys resident procedures.

Writing Procedures

BASIC09 is a *modular* programming language. Several procedures can occupy memory at the same time. Each procedure performs a particular function but can also interact with others to form a sophisticated program.

To create or change procedures, enter the *edit mode* by typing either `edit` [ENTER] or [E] [ENTER] at the `B:` prompt. From now on, when directing you to enter the edit mode, this manual uses the easier to type [E] command.

Each time you type a procedure line and press [ENTER], the editor checks for common errors. This automatic checking lets you catch mistakes before you run the program, saving you testing and rewriting time. You can even let the automatic checking help you learn the rules of BASIC09. If you are not sure about a syntax, go ahead and type it the way you think is correct. If you guess wrong, BASIC09 shows where the error is and displays a message to tell what is wrong.

BASIC09's use of modules lets you divide large and complex projects into smaller, easily manageable sections. Not only are the smaller procedures easier to write and understand, they are also easier to test. As well, because BASIC09 lets you *call* procedures that are outside the *workspace* (the computer's memory where you write and edit procedures), you can accumulate libraries of procedures to incorporate into future programs.

You can work on a program's procedures either individually or as a group. For example, to work on the procedures as a group, save your workspace procedures into a single disk file. When you subsequently load the file, BASIC09 automatically loads all of the procedures.

Modules of Other Languages

BASIC09 can incorporate procedures from other languages, such as Pascal, C, or assembly language. Several users can then share the procedures.

Executing Procedures

You execute or *run* programs from the command mode. When you enter a procedure, BASIC09 compiles it. This means that the procedure is ready for execution as soon as you exit the edit mode. For instance, if you create a program named Greeting, you can execute it by typing from the command mode:

`run greeting` [ENTER]

Leaving BASIC09

There are two ways you can exit from BASIC09:

- At the `B:` prompt, type:

  `bye` [ENTER]

- Or, at the `B:` prompt, press [CTRL][BREAK].

When you use either method, the OS-9 prompt appears immediately indicating that the operating system is waiting for a command.

Note: When you exit BASIC09, you lose all procedures residing in the workspace. Be sure to save them on disk before leaving BASIC09.

The Keyboard and BASIC09

You can use some keys and *key sequences* to produce special characters and to accomplish special BASIC09 functions. You initiate a key sequence by pressing one key and holding it down while pressing a second key. The following list summarizes your keyboard's special functions:

ALT	Produces graphic characters. Press ALT *char* where *char* is a keyboard character).
CTRL	A control key that you use with other keys. (See below.)
BREAK or CTRL E	Stops the current program execution and returns to the `B:` prompt in BASIC09's command mode.
← or CTRL H	Moves the cursor back one space.
CTRL _	Produces an underscore character.
CTRL ,	Produces a left brace ({).
CTRL .	Produces a right brace (}).
CTRL #	Produces a tilde (~).
CTRL /	Produces a backslash (\).
CTRL BREAK	Performs an ESCAPE function and sends an end-of-file message to a program receiving keyboard input. To be recognized, CTRL BREAK must be the first thing typed on a line.
SHIFT BREAK	Stops execution of a program and causes BASIC09 to enter the Debug mode.
CLEAR	Displays the next window.
SHIFT CLEAR	Displays the previous window.
SHIFT ← or CTRL X	Deletes the current line.
CTRL 0	Activates or deactivates the shift lock function.
CTRL 1	Produces a vertical bar (|).
CTRL 7	Produces an up arrow (↑).
CTRL 8	Produces a left bracket ([).
CTRL 9	Produces a right bracket (]).

CTRL A	Redisplays the last line typed, and positions the cursor at the end of the line, but does not process the line. Press ENTER to process the line, or edit the line by backspacing. If you edit, press CTRL A again to display the edited line.
CTRL D	Redisplays the current command line.
CTRL W	Temporarily halts video output. Press the space bar to resume output.
ENTER	Performs a carriage return or executes the current command line.

Chapter 2

Sample Session

Although BASIC09 has several functions or modes, they all work together to make programming as simple as possible. The easiest way to learn how BASIC09 and its functions operate is to write and run a program. This chapter provides sample statements and instructions to help you learn how to use BASIC09.

To create and execute a program:

1. Load BASIC09 and enter the edit mode.
2. Type the BASIC program.
3. Enter the system mode and test the program's execution.
4. Debug the program. (Correct any programming errors.)
5. Save the completed program on disk.
6. Load the program into memory and use it.

To begin the program, execute BASIC09. To be sure you have enough room in which to work, reserve a workspace of 10,000 bytes by typing:

```
basic09 #10K [ENTER]
```

The BASIC09 system mode prompt, `B:`, appears after the copyright message. In the system mode, you can do such things as save and load procedures, change workspace size, and rename and delete procedures.

Creating a Procedure

To write procedures, you must be in the edit mode. You get there by typing:

```
e [ENTER]
```

This causes the screen prompt to change to `E:`, and the screen displays:

```
PROCEDURE Program
```

Because you didn't give a program name when you entered `e`, BASIC09 selects the name Program for you. Now, you must write the code to make Program do something.

Commands and Program Lines

There are two responses you can give at the edit mode prompt. You can type an edit command, or you can type a program line. If you type a program line, **you must type a space as the first character in the line.** If you type an edit command, do **not** precede it with a space. To make listings easier to read, this manual uses the symbol ▯ to indicate spaces before every line. It also uses the ▯ symbol in some procedure lines to indicate the correct number of spaces needed. Whenever you see either a space or a ▯ symbol in a procedure you are typing, press the space bar.

To type the procedure in this chapter, begin each line at the `E:` prompt. After typing a line, check it for mistakes. If you make a mistake, use [←] to move the cursor back. Correct the mistake. Then, type the remaining portion of the line. If there are no mistakes, press [ENTER].

BASIC09 checks each line when you press [ENTER]. If you make a mistake in syntax or form, BASIC09 displays an error message. An arrow points to the place in the program line where the error occurred, and a message number indicates the type of error. Refer to Appendix A for an explanation of the error codes.

If, after you enter a line, you find that you made a mistake, type `d` [ENTER] to delete the line. Then, retype the line. Later, the manual tells you how to change text in existing lines.

The following program helps you do a bit of arithmetic. To get a feel for BASIC09, type and execute the program as directed. Remember, when you see either a space or ▯, press the space bar.

```
▯DIM NUMBER1,NUMBER2:INTEGER
▯INPUT "Type Number...";NUMBER1
▯INPUT "Type another....";NUMBER2
▯PRINT "The sum of the numbers is... ";
▯PRINT NUMBER1 + NUMBER2
▯END
```

Executing a Procedure

To execute the procedure, quit the edit mode by typing `q` [ENTER]. The compiler further processes your procedure, and the `B:` prompt reappears. To execute the program, type:

```
run [ENTER]
```

Type in numbers when asked, and the procedure produces their sum. If you want to save the program on disk, the next chapter tells you how. Chapter 4 introduces several other edit mode commands to search, display, insert, and change programs lines and text.

The System Mode

The BASIC09 *command interpreter* processes system commands. At the `B:` prompt, you can enter system commands in either upper- or lowercase letters. Some commands operate on the procedures in the workspace. Others provide functions independent of any procedures. Following is a list of all system commands and their purposes.

Command	Function
$	Calls the shell command interpreter to execute an OS-9 command.
BYE or [CTRL] [BREAK]	Returns you to the OS-9 system or to the program that called BASIC09.
CHD	Changes the current OS-9 data directory.
CHX	Changes the current OS-9 execution directory.
[ENTER] **or DIR**	Displays the name, size, and variable storage requirement of each procedure in the workspace.
EDIT or E	Enters the procedure editor-compiler mode.
KILL	Erases one or more procedures from the workspace.
LIST	Displays a formatted listing of one or more procedures.
LOAD	Loads all procedures from a disk file into the workspace.
MEM	Displays in bytes the current workspace size, or reserves a specified amount of memory for the workspace.
PACK	Condenses (compiles) one or more procedures.
RENAME	Changes a procedure's name.
RUN	Causes a procedure in the workspace to execute.
SAVE	Writes one or more procedures to disk.

Renaming Procedures

BASIC09's RENAME function is important for two reasons: First, it lets you load into the workspace procedures that have the same name. After you rename the workspace procedure you can load the second file. Second, if you let BASIC09 use the default procedure name, "Program," you can rename the procedure before saving it to disk. By doing this, you avoid writing over—and destroying—an existing procedure file.

To change the name of the procedure you created in the previous chapter from Program to Add, type:

```
rename program add [ENTER]
```

Listing Procedure Names

You can use the DIR command to see if RENAME worked properly. DIR displays the names and sizes of all procedures in memory. Because programmers use this command frequently, the system recognizes a shorthand call. Instead of typing `dir` [ENTER], you only need to press [ENTER]. This displays a table of the procedures in the following format:

```
  Name                Proc-Size        Data-Size
  *add                   182              32
   add1                  217              42
   add2                  218              42
 2198 free
```

Proc-Size refers to the number of memory bytes required for the procedure. *Data-Size* refers to the number of memory bytes required for the procedure's variables and data structures. The asterisk indicates the current procedure. System commands act on the current procedure unless you indicate otherwise.

The last line of the DIR display tells you how many free bytes of memory remain in the BASIC09 workspace.

Listing Procedures

You can use the LIST command to view procedure lines. To display the current procedure, type:

```
list [ENTER]
```

For example, this is the listing of a procedure named Alpha.bak:

```
PROCEDURE Alpha_bak
 0000      DIM A:STRING
 0007      DIM T:INTEGER
 000E
 000F      PRINT "Here is the alphabet
           backwards:"
 0032      PRINT
 0034      FOR T=90 TO 65 STEP -1
 004A        PRINT CHR$(T);
 0051        PRINT " ";
 0057      NEXT T
 0062      PRINT
 0064      PRINT
 0066      END
```

When you list a BASIC09 procedure, the system precedes each line with a *relative* storage address. The relative address of the first procedure line is always 0. In the previous example, the beginning address of the second procedure line in the workspace is 07 units from the beginning. The beginning address of the third line is 0E hexadecimal (14 decimal) storage units from the procedure beginning.

These I-Code addresses provide a way for the compiler to let you know where it finds an error when one occurs.

Because BASIC09 compiles programs into I-Code, it must *disassemble* them before it can display them on the screen. This means that the lines might not look exactly as typed. For instance, BASIC09 converts lowercase *keywords* (command names) to uppercase. BASIC09 also eliminates some spaces. If your program uses control statements such as IF/THEN, FOR/NEXT, and LOOP/ENDLOOP, the lines in these decision making or looping structures are indented as shown in the Alpha.bak example. Regardless of the appearance of your listed procedures, they execute correctly if you type their commands correctly.

Listing Procedures to a File

There might be times when you want to send a formatted procedure listing, including I-Code addresses, directly to a file. You can do this using OS-9's redirection symbol, >. To save the Alpha.bak procedure on a file named Alpha.list in the current data directory, type:

```
list alpha.bak >alpha.list [ENTER]
```

If you have several procedures in the workspace and want to list more than one to a disk file, separate the procedure names with commas, like this:

```
list alpha.one,alpha.two,alpha.three >alpha.all
[ENTER]
```

In both of the preceding cases, the system creates the Alpha.list file and stores the specified listings in it. If you use a file name that already exists, BASIC09 displays the prompt:

```
Rewrite?:
```

If you press [Y], the system destroys the original file and overwrites it with the new listing. If you press [N], the LIST process terminates.

If you wish to list a procedure, or group of procedures, to a file that is not in the current data directory, be sure to specify the complete pathlist, such as:

```
list alpha.bak > /d1/programs/alpha.rev [ENTER]
```

Listing Procedures to a Printer

In the same manner as you list procedures to a disk file, you can list one or more procedures to your printer. Make certain your printer is connected and turned on, then again use the redirection symbol, but this time specify the printer device, like this:

```
list alpha.bak >/p [ENTER]
```

Or:

```
list alpha.one,alpha.two,alpha.three >/p [ENTER]
```

Using a Wildcard

Using the OS-9 wildcard, *, you can list all procedures in the workspace. For instance, if the procedures Alpha.one, Alpha.two, and Alpha.three exist, list them to the screen by typing:

```
list* [ENTER]
```

Send the list to a file by typing:

```
list* alpha.all [ENTER]
```

Or send the list to your printer by typing:

```
list* /p
```

Note: When you use the wildcard, the name of the file or device to receive the listing immediately follows the LIST* command. Do not use the redirection symbol.

Saving Procedures

You can save one or more procedures to disk using the SAVE command. Unlike LIST, SAVE does not include relative addresses. However, the syntaxes for the SAVE and LIST commands are identical. To save the procedure Alpha.bak to the current data directory, type:

```
save alpha.bak alpha.bak [ENTER]
```

If Alpha.bak is the current procedure, you can save it in a file named Alpha.bak by typing `save` [ENTER].

To save all of the procedures in the workspace to a file named All.programs in the current data directory, type:

```
save* All.programs [ENTER]
```

As with LIST, to save one or more procedures in a file that is not in the current data directory, make sure you specify a complete pathlist.

To save all the files in the workspace to a disk file with the same name as the current procedure, type `save*` [ENTER].

If the disk file you specify does not exist, BASIC09 creates it. If it does exist, the system displays the prompt:

```
Rewrite?:
```

Press [Y] to write over the old file with the specified file. The old file is destroyed.

Press [N] to terminate the SAVE operation.

Loading Procedures

To load a saved procedure back into BASIC09's workspace, use the LOAD command and specify the appropriate pathlist. For instance, if your current directory is still the directory containing Alpha.bak, load the procedure by typing:

```
load alpha.bak [ENTER]
```

To load Alpha.bak from the PROGRAMS directory on Drive /D1, type:

```
load /d1/programs/alpha.rev [ENTER]
```

You can run and edit a loaded procedure in exactly the same manner as you would a procedure you created.

You can load any number of procedures into the workspace as long as your computer has sufficient memory. However, be careful that you do not load a procedure with the same name as a procedure already existing in the workspace. If you do, the new procedure overwrites (destroys) the original procedure. You can rename workspace procedures to avoid this problem.

Deleting Procedures from the Workspace

You can clear the workspace of one or more procedures using the KILL command. For instance, to remove Alpha.bak from the workspace, type:

```
kill alpha.bak [ENTER]
```

To remove more than one procedure from the workspace, separate the procedure names with commas. To delete Alpha.one and Alpha.two, type:

```
kill alpha.one,alpha.two [ENTER]
```

To clear the entire workspace, regardless of the number of procedures it contains, use the BASIC09 wildcard, *. Type:

```
kill* [ENTER]
```

Changing Directories

You change working directories in BASIC09 and OS-9 in the same manner, by using the CHD and CHX commands. CHD changes the data directory, and CHX changes the execution directory.

BASIC09 saves files in, or loads files from, the data directory, unless you specify differently in the command pathlist. It stores packed procedures in, or loads PACKed procedures from, the execution directory, unless you specify differently in the command's pathlist.

Also, if you want to access OS-9 commands from BASIC09, the system first looks for the commands in memory. If they are not there, it looks for them in the execution directory, unless you specify differently.

If your data directory is the ROOT directory, and you wish to change to a directory named PROGRAMS that is a subdirectory of the ROOT directory, type the following command from the command mode `B:` prompt

```
chd programs [ENTER]
```

If your current execution directory is the system's CMDS directory, and you want to change to a CMDS directory in the subdirectory BASIC, type:

```
chx basic/cmds [ENTER]
```

Whenever you change to a directory other than an immediate subdirectory, specify a complete pathlist.

Executing OS-9 Commands

BASIC09 lets you use OS-9 commands at any time from the system mode. To do so, precede the command with a dollar sign ($). For instance, to look at the current data directory, type:

```
$dir [ENTER]
```

To view the current execution directory, type:

```
$dir x [ENTER]
```

All OS-9 commands are available, and you can copy files, format diskettes, list files, or use any other functions from the system mode. The only restriction is that your computer must have enough free memory to handle the command you call. If you find that there is not enough memory, try using the MEM command to reduce reserved memory. Then, try the command again.

Auto-Execute Procedures

The BASIC09 compiler makes two passes through the procedures you write. When you enter the command, the compiler performs an initial compilation, checking for any syntax errors. When you leave the edit mode, the system compiles the procedure a second time and checks for any programming errors. With the PACK command, you can further compile your procedures so that they are smaller and execute even faster.

PACK causes an extra compiler pass that removes names, line numbers, and non-executable statements. **Before packing a procedure, be sure you save it. Unless you do so, you cannot make further changes to the procedure.**

Once you pack a file, you cannot list or edit the packed version. However, if you save the procedure to disk before packing, you can still list and edit the original file, then pack it again.

When you save a packed procedure on disk, BASIC09 does not normally store it in the data directory. Because the procedure is now *executable*, the system stores it in the current execution directory.

For instance, to convert Alpha.bak to a packed procedure in the execution directory, type:

```
pack alpha.bak [ENTER]
```

If you want to save a packed procedure under a different filename, use the OS-9 redirection symbol:

```
pack alpha.bak > backwards [ENTER]
```

After packing a procedure, you can delete it from the workspace. If you then run it, BASIC09 automatically loads the file from disk and executes it.

The following is a sequence of commands that demonstrate packing and executing a procedure named Alpha.bak:

`pack alpha.bak` [ENTER]	packs the procedure and stores it in the execution directory.
`kill alpha.bak` [ENTER]	deletes the procedure from the workspace.
`run alpha.bak` [ENTER]	loads the file into memory *outside* the workspace and executes it.
`kill alpha.bak` [ENTER]	deletes the module from memory

You do not need to kill the file immediately after execution, but until you do, the file reduces available memory.

The Edit Mode

You briefly used the BASIC09 built-in editor to create the Add procedure in Chapter 2. In addition to the features you learned there, the editor has other important functions.

Although you can use any text editor or word processor to write BASIC09 procedures, the BASIC09 editor offers two handy features:

- It is both *string* and *line number* oriented. You can search for strings of characters, and replace them, and you can reference text with optional line numbers.
- It interfaces with the compiler and *decompiler*. This feature lets BASIC09 check continuously for syntax errors and enables you to use procedures that conserve memory.

Edit Commands

The following is a summary of the edit commands:

Command	Function
ENTER	Moves the edit pointer to the next line. Causes a command to execute.
+*number*	Moves the edit pointer ahead *number* lines.
+*	Moves the edit pointer to the last line.
-*number*	Moves the edit pointer back *number* lines.
-*	Moves the edit pointer to the first line.
text	Inserts an unnumbered text line before the current line.
ntext	Inserts the line numbered *n* in its correct numeric position.
n	Moves the edit pointer to the line numbered *n*.
c/*str1*/*str2*/	Changes the next occurrence of *str1* to *str2*.

Command	Function
c*/*str1/str2*/	Changes all occurrences of *str1* to *str2*.
d	Deletes the current line.
d*	Deletes all the lines in the procedure.
l	Lists the current procedure line.
l*	Lists all the procedure lines.
q	Terminates the edit session.
r	Renumbers lines beginning at the current line in increments of 10.
r*	Renumbers all lines in increments of 10.
r *n*	Renumbers lines beginning at Line *n* in increments of 10.
r *n1 n2*	Renumbers lines beginning at Line *n1* in increments of *n2*.
s /*string*/	Searches for the first occurrence of *string*.
s* /*string*/	Searches for all occurrences of *string*.

Using the Editor

The easiest way to understand the edit commands is to use them. The following sections show you the functions of BASIC09 edit mode.

The manual uses line numbers in the following procedure to acquaint you with all the functions of the editor. Remember, however, line numbers are not required with BASIC09. Procedures and programs without line numbers are shorter, faster, and easier to read.

First, you need a procedure with which to work. Position yourself in the system mode. Then, type this line:

```
e prose [ENTER]
```

Now, type the following. (Remember, the small rectangle represents a space.)

```
□100 DIM PHRASES(30):STRING
□120 FOR T=1 TO 30
□130 READ PHRASES(t)
□140 NEXT T
□160 PRINT
□170 FIRST=RND(10)
□180 SECOND=RND(9)+11
□190 THIRD=RND(9)+21
□200 PRINT PHRASES(FIRST);
□210 PRINT PHRASES(SECOND);
□220 PRINT PHRASES(THIRD);
□240 PRINT
□300 DATA "Love□","An orange□",
"Humanity□","A kiss□"
□310 DATA "A dark cloud□","A goose feather□",
"A Popsicle□"
□320 DATA "Home cooking□","Cold pizza□",
"Rock n' Roll□"
□330 DATA "is charming like□","makes me dream of□"
□340 DATA "is as sticky as□","can ooze
like□","smells like□"
□350 DATA "can be as tough to forget as□","can
hurt like□"
□360 DATA "can be as cynical as□","makes a mockery
of□"
□370 DATA "drives me as crazy as□"
□380 DATA "a sticky lollipop.","a web of
intrigue."
□390 DATA "castor oil.","a chocolate bath.","a
broken toe."
□400 DATA "honey and things.","personal
defeat.","a wet diaper."
□410 DATA "strange happenings.","a pennyless
purse."
```

When you finish typing the procedure, type q [ENTER] to return to the system mode. Now you can test the program by typing either:

```
run [ENTER]
```

or

```
run prose [ENTER]
```

After trying the procedure, return to the edit mode by typing `e` [ENTER].

After displaying the procedure's name, the editor displays Line 100 preceded by an asterisk. The asterisk lets you know which line is the *current line* (or the line at which the edit pointer is located).

Searching Through a Procedure

You can examine a procedure in three ways:

- Press [ENTER] to display the procedure one line at a time.
- Skip through the procedure to a particular line.
- List part or all of the procedure to the screen.

When you use either of the first two methods, the line you select to display becomes your current line. When you use the third method, the current line does not change.

Using [ENTER]

If you are still positioned at Line 100, but want to examine the first line of data, Line 300, press [ENTER] 12 times to move down.

Using the Plus Sign to Move Forward

Another method of moving to a specific line is to type a plus sign followed by the number of lines you need to advance to get there. Positioned at Line 100, you can type:

```
+12 [ENTER]
```

Whether you press [ENTER] or use the plus sign, the last line displayed is now your current line.

Accessing a Line Using the Line Number

The third way to move to a particular line is to type the line number, followed by [ENTER]. For instance, to jump back to Line 100, type:

    100 [ENTER]

The editor displays Line 100 and makes it your current line.

Using the Minus Sign to Move Backward

In the same manner that you move forward in the procedure using the plus sign, you can move backward using the minus sign, or hyphen.

Type 300 [ENTER] to return to Line 300. To display Line 240 and make it your current line, type:

    - [ENTER]

To display Line 190 and make it your current line, type:

    -4 [ENTER]

The Global Symbol

The BASIC09 editor also makes use of the asterisk as a global symbol. For instance, following a command with an asterisk causes that command to affect the entire procedure.

This feature lets you move quickly to the beginning and end of the procedure. To return to Line 100, the first line, type:

    -* [ENTER]

To move to the end of the procedure, past all the numbered lines, type:

    +* [ENTER]

Using LIST

The LIST command lets you select one or more lines for display on your screen. To see this, make the first line your current line, then type:

```
1 [ENTER]
```

To list one or more lines, type the LIST command followed by the number of lines you want displayed. For instance, typing l5 [ENTER] causes the current line and four others to appear on the screen, as shown in the following sequence of commands and the resulting display:

```
-* [ENTER]
l5 [ENTER]
PROCEDURE Prose
100 DIM PHRASES(30): STRING
120 FOR T=1 TO 30
130READ PHRASES(T)
140 NEXT T
160 PRINT
```

You can also use LIST with the BASIC09 global symbol, *. Typing an asterisk after the LIST command produces a listing of the entire procedure.

Deleting Lines

Earlier, the manual showed that you can delete the current line by typing d [ENTER]. Because this is such a simple process, be sure you don't do it by accident. Removing the wrong line, or too many lines, is very frustrating in a complex procedure.

You can also remove a group of lines from a procedure by typing d, followed by the number of lines you want to delete. This command deletes the current line and specified following lines. Again, be careful.

You can remove all of the lines in a procedure by using the global symbol, *. Typing d* [ENTER] erases all procedure text. However, the procedure name still resides in the workspace. To delete an entire procedure, including the name, use the KILL command from the system mode.

If you decide you don't like the nouns used in the DATA lines of the Prose procedure, erase all of the DATA lines containing nouns (Lines 300-320) and replace them. To do so, make Line 300 your current line by typing:

```
300 [ENTER]
```

Then type:

```
d [ENTER]
```

Line 300 disappears and Line 310 takes its place as the current line.

An alternate method of deleting the DATA lines uses only one command. To delete Lines 300 through 410, follow the DELETE command with the number of lines you want to remove—in this case, three:

```
d3 [ENTER]
```

Lines 300, 310, and 320 disappear. Line 330 becomes the current line. Move back a line to check that the deletions worked. The line numbers now skip from 240 to 330.

Now, you need new nouns for the procedure. Type them in the same style as the old lines, such as:

```
□300 DATA "A Telephone□,"A tickle□",
"A girl□","A boy□"
□315 DATA "Bad luck□","Money□","A bad bet□",
"A lumpy bed□"
□320 DATA "A deep thought□","Sunlight□"
```

Save a copy of your procedure to disk by exiting the editor and using the SAVE command. Then return to the edit mode and try the global delete by typing:

```
d* [ENTER]
```

Changing Text

Using CHANGE tells the editor to search for existing text and replace it with new text. CHANGE, like DELETE, can easily cause unwanted results if you are not careful.

The CHANGE command requires that you use *delimiters* to separate the command from the search text, and to separate the search text from the new text. You can select any of the following characters for a delimiter, as long as it does not appear in either the search text or the new text:

! # % ^ & ( ) - + = { } [ ] " " < > , . ? / \ |

Do not use the global symbol (*) for search and replace operations. This manual uses a slash (/) as the CHANGE delimiter.

The following steps outline the correct use of CHANGE:

1. Position the editor either before or on the line in which you want to make a change.
2. Type `c` (for CHANGE). Do not use a preceding space.
3. Type a delimiter character, such as `/`.
4. Type the characters to be changed, following them with the delimiter.
5. Type the new text, followed by the delimiter.
6. Press [ENTER].

Note: It is a good idea to turn on OS-9's upper- and lowercase function before attempting change or search operations. If you do not, you cannot tell whether the text you want to find is upper- or lowercase, or some combination of the two. If you type the wrong case, the change or search fails.

In case you didn't notice when typing the procedure, Line 410 contains an incorrectly spelled word, pennyless. To correct this error, type the following:

```
c/pennyless/penniless/ [ENTER]
```

Immediately, the editor displays Line 410, with pennyless changed to penniless.

Suppose you decide to change the number of sentence combinations available in Prose. The procedure now has 30 data entries. If you add five subjects, five verb phrases, and five objects, the procedure also needs other changes (for instance, the DIM statement in Line 100, the loop size in Line 120, and the RND statements in Lines 170 through 190).

A quick way to change the number 30 in Lines 100 and 120 is to use CHANGE's global function. To change all occurrences of 30 to 45, position the editor at Line 100, and type:

```
c*/30/45/ ENTER
```

Use the CHANGE and global CHANGE functions to adjust the RND statement values in Lines 170, 180, and 190.

As well as making changes, you can use the CHANGE command to quickly delete portions of text within a line. To do this, type delimiters without new text, in this fashion:

```
c/□feather// ENTER
```

This command changes the text `A goose feather` in Line 210 to `A goose`.

Searching for Text

The editor's SEARCH command, S, works in the same manner as the CHANGE command. However, SEARCH only requires you to specify a block of text to find.

With SEARCH, you use delimiters to enclose the text to find. To test the function, position the editor at the beginning of text by typing:

```
-* ENTER
```

Now, search for the word phrases, by typing:

```
s/phrases/ ENTER
```

The screen displays:

```
*0000 100 DIM phrases(30):STRING
```

To find all occurrences of phrases throughout the procedure, use the global symbol. Type:

```
s*/phrases/ ENTER
```

Renumbering Lines

The RENUMBER command, R, reorders all numbered lines and all references to numbered lines. You can give RENUMBER either one or two parameters. The first is the beginning line number. The second is the increment you want. The default increment is 10.

For instance, the Prose procedure line numbers skip from Line 100 to Line 120. You can renumber the entire procedure by moving the editor to Line 100, and then typing:

```
r 10 [ENTER]
```

To change the numbering to increments of 5, beginning at Line 100, type:

```
r 100,5 [ENTER]
```

You can also change line numbering in portions of the procedure. To do this move the editor to the line where you want the new numbering to begin. Then, type in the new parameters. To renumber Line 100 as Line 200 and continue with increments of 10, position the editor at Line 100. Then, type:

```
r 200,10 [ENTER]
```

If you are not positioned at the first line of a procedure, but you wish to renumber all lines, you can use the global symbol to do the job. From anywhere in the procedure, type:

```
r* 100,10 [ENTER]
```

This renumbers the entire procedure in increments of 10.

Adding Lines

There are two ways to add new lines to a procedure. You can:

- Position the editor one line below the position for the new line. Then, type the new line and press [ENTER]. When inserting lines without numbers, be sure to type a space as the first character of the line to tell the editor that the following text is a new procedure line.
- Type a new line, giving it a line number that falls between two existing line numbers.

The following procedure adds more choices to the Prose program. It also adds a feature that lets you press ENTER for additional output, rather than having to rerun the procedure. Use the information presented in this section to help you insert the new lines into your program. Because you must change some lines, as well as add lines, the following listing includes the entire procedure.

Referring to the original Prose listing, the lines to change are: 100, 120, 170, 180, and 190.

The lines to add are: 110, 150, 230, 250, 260, 270, 305, 325, 372, 374, 376, 420, 430.

```
PROCEDURE prose2
100 DIM PHRASES(45):STRING
110 DIM RESPONSE:STRING
120 FOR T=1 TO 45
130 READ PHRASES(t)
140 NEXT T
150 REPEAT
160 PRINT
170 FIRST=RND(15)
180 SECOND=RND(14)+16
190 THIRD=RND(14)+31
200 PRINT PHRASES(FIRST);
210 PRINT PHRASES(SECOND);
220 PRINT PHRASES(THIRD);
230 PRINT
240 PRINT
250 PRINT "□□□□□□□□□□□Press ENTER for another
witticism..."
260 INPUT "□□□□□□□□□□□Or press the SPACEBAR and press
ENTER to end...",RESPONSE
270 UNTIL RESPONSE>""
300 DATA "Love□","An orange□","Humanity□",
"A kiss□"
305 DATA "A computer□","A book□","Misery□"
310 DATA "A dark cloud□","A goose feather□",
"A Popsicle□"
320 DATA "Home cooking□","Cold pizza□",
"Rock n' Roll□"
325 DATA "Snow in June□","A glass house□"
330 DATA "is charming like□","makes me dream of□"
```

```
340 DATA "is as sticky as□","can ooze like□",
"smells like□"
350 DATA "can be as tough to forget as□",
"can hurt like□"
360 DATA "can be as cynical as□",
"makes a mockery of□"
370 DATA "drives me as crazy as□"
372 DATA "can bother me like□","blackens my hopes
like□"
374 DATA "can tickle me like□","can be as funny
as□"
376 DATA "has the effect of□"
380 DATA "a sticky lollypop.","a web of
intrigue."
390 DATA "castor oil.","a chocolate bath.","a
broken toe."
400 DATA "honey and things.","personal
defeat.","a wet diaper."
410 DATA "strange happenings.","a penniless
purse."
420 DATA "a slimy snake.","a bad habit."
430 DATA "a bad memory chip.","a good fight.","a
silly friend."
```

The Next Step

Even the best programmers make mistakes—a lot of them. BASIC09 provides a way to catch programming mistakes quickly and correct them. The next chapter tells you about BASIC09's powerful debugging functions.

Chapter 5

The Debug Mode

The term *debug* refers to the process of finding programming errors and correcting them. BASIC09's debugging features include *symbolic* debugging capabilities that let you examine variable values and test and manipulate procedures.

With Debug, you can:

- Examine and change variables.
- Trace procedure execution. Debug lets you execute procedures and watch them run in slow motion.
- Pause procedure execution.
- Resume procedure execution.
- Set procedure *breakpoints* that automatically switch to the debug mode.
- Select the use of degrees or radians for trigonometric functions.
- Perform calculations.
- Call OS-9 system commands.

Entering the Debug Mode

You enter Debug:

- Automatically, whenever an error occurs during the execution of a procedure (unless you have included an ON ERROR GOTO statement to handle the error).
- Automatically, when a procedure executes a PAUSE statement.
- When you press CTRL C during the execution of a procedure.

You can tell when BASIC09 enters the Debug mode by the appearance of the `D:` prompt. When you see `D:`, followed by the cursor, Debug is waiting for your command.

The following is a reference of all the Debug commands and what they accomplish:

Command	Function
$	Calls OS-9's command shell interpreter to run a program or an OS-9 command. From the Debug prompt, type `$`, followed by the name of the program or command you want to execute. Example: `$list procedure_one` [ENTER]
BREAK	Sets a breakpoint immediately before the specified procedure. Use this command to re-enter Debug when one procedure calls another. If you have three procedures that call each other—Proc1, Proc2, and Proc3—and Proc3 does not seem to pass the correct values to Proc2 when it returns, set a breakpoint at Proc2. This causes BASIC09 to enter Debug before re-entering Proc2. You can then check your variable values. You can use one breakpoint for each active procedure. Debug removes breakpoints immediately after encountering them. A procedure must run before you can set a breakpoint in it. Use BREAK to stop execution when a called procedure returns to a procedure previously executed. Example: `BREAK proc2` [ENTER]
CONT	Causes procedure execution to continue. Example: `cont` [ENTER]
DEG/RAD	Selects either degrees or radians as the unit of measurement for trigonometric functions. DEG and RAD affect only the current procedure. Examples: `deg` [ENTER] `rad` [ENTER]

DIR

Displays the name, size, and variable storage requirements of each procedure in the workspace. The current working procedure has an asterisk before its name. All packed procedures have a dash before their names. DIR also shows the available memory in the workspace.

If you provide a pathlist, DIR sends its data to the specified file.

Example: `dir` [ENTER]
`dir procedures` [ENTER]

Q

Terminates execution of the procedure, closes any open paths, and exits to the System mode.

Example: `q` [ENTER]

LET

Assigns a new value to a variable. You must specify variable names that are already initialized by your program. In the Debug mode, you cannot use LET to copy one array to another array as you can in BASIC procedures.

Example: `let a := 0` [ENTER]
`let fruit := "oranges"` [ENTER]

LIST

Displays a source listing of the suspended procedure. The display is formatted and includes I-code addresses. An asterisk appears to the left of the last executed statement.

Example: `list` [ENTER]

PRINT

Displays the values of variables used in the suspended procedure. You cannot introduce new variable names in the Debug mode, and you cannot display array structures.

Example: `print fruit` [ENTER]

STATE

Lists the *nesting* order of active procedures. STATE displays the highest-level procedure at the bottom of the calling list. The lowest-level procedure is the suspended procedure.

Example: `state` [ENTER]

STEP Causes execution of the suspended procedure in specified increments. For example, typing `STEP 5` [ENTER] executes the equivalent of the next five statements. If you enter STEP without an increment value, the step rate is 1.

Using STEP with the trace function lets you observe the source lines as they execute.

Because compiled I-code contains actual statement memory addresses, the *top* or *bottom* statements of loop structures execute only once. For example, in FOR/NEXT loops, FOR executes once, and the statement following FOR appears to be the top of the loop.

TRON/TROFF Turns on or turns off the trace function. Trace on (TRON) causes the system to reconstruct the compiled code of each statement line into source code. Debug displays the source code before the statement is executed. If the statement causes the evaluation of one or more expressions, Debug displays each result following the statement. The result is preceded by an equal sign.

The trace function is local to the current procedure. If the suspended procedure calls another procedure, Debug suspends the trace function until control returns to the original procedure. However, once you turn on trace for a procedure, it continues in effect until you turn it off. This means that if you turn trace on in a called procedure, and another procedure subsequently calls it, trace continues to display the called procedure's operations.

Example: `tron` [ENTER]
`troff` [ENTER]

When Things Go Wrong

Programming errors show up in two ways. Either your procedure produces incorrect results, or it terminates prematurely.

In the first instance, you can stop your procedure and enter Debug by pressing [CTRL][C].

However, sometimes your program executes too quickly to allow you to stop it at the appropriate place. In this case, you can use the Edit mode to insert a PAUSE command where you wish the procedure to stop. PAUSE causes the procedure to halt execution and enter the Debug mode.

Once in Debug, you can use the PRINT command to examine the procedure variables. You can use LET to manipulate the variable values to determine where the error or errors occur. Perhaps you forgot to initialize a variable or forgot to increase a loop counter.

Using the Trace Function

Sometimes, errors are more difficult to discover. If so, the next step is to use the trace function. To do this, type:

```
tron [ENTER]
```

Now press [ENTER]. Each time you press [ENTER], Debug executes one line of the procedure. You can see the original source statement, and if an expression is evaluated, Debug prints the result of the expression, preceded by an equal sign.

In this manner, you can step through the entire procedure, or any part of it, examining variable values as you go.

What About Loops?

The STEP command is helpful if you find yourself tracing the operation of a loop. Once you determine that the loop works correctly, you can avoid tedious, step-by-step repetitions by turning trace off and using STEP to quickly run through the loop. Then, turn trace back on and resume single-step debugging. For instance, type:

```
troff [ENTER]
step 200 [ENTER]
tron [ENTER]
```

In Multiple Procedures

Although the trace function is local to a procedure, you can pause and turn on the trace function in as many procedures as you wish. Trace continues to operate in each procedure until you turn it off using TROFF.

To cause a procedure to halt execution when it is called by another procedure, use the BREAK command.

Data and Variables

Data Types

Data is information on which a computer performs its operations. Data is always numeric but, depending on your computer application, it can represent values, symbols, or alphabetic characters. This means that the same items of *physical* data can have very different *logical* meanings, depending on how a program interprets it.

For instance, 65 can represent:

- A numeric value to be used in a calculation.
- The location of a memory address.
- The *offset* of a memory location.
- The two character symbols 6 and 5.
- The character A in the ASCII table. ASCII is the abbreviation for the American Standard Code for Information Interchange.

Because of the differences in how BASIC09 uses data, the system lets you define five types of data. For instance, there are three ways to represent numbers. Each has its own advantages and disadvantages. The decision to use one way or another depends on the specific program you are developing. The five BASIC09 data types are byte, integer, real, string, and Boolean.

In addition to the preceding data types, there are *complex data types* you can define. The manual discusses complex data structures at the end of this chapter.

The *byte*, *integer*, and *real* data types represent numbers.

The *string* data type represents character data (alphabet, punctuation, numeric characters, and other symbols). The default length of strings is 32 characters. Using the DIM statement, you can specify strings of both longer and shorter lengths.

The *Boolean* data type represents the logical value, TRUE or FALSE.

You can create arrays (lists) of any of these data types with one, two, or three dimensions. The following table shows the data types and their characteristics:

Type	Allowable Values	Memory Requirements
BYTE	Whole numbers (0 to 255)	One byte
INTEGER	Whole numbers (-32768 to 32767)	Two bytes
REAL	Floating point ($\pm 1*10^{\wedge \pm 38}$)	Five bytes
STRING	Letters, digits, punctuation	One byte per character
BOOLEAN	True or false	One byte

Real numbers appear to be the most versatile. They have the greatest range and are floating point. However, arithmetic operations involving real numbers execute much more slowly than those involving integer or byte values. Real numbers also take up considerably more memory storage space than the other two numeric data types.

Arithmetic involving byte values is not appreciably faster than arithmetic involving integers, but byte data conserves memory.

If you do not specify the type of variable (a symbolic name for a value) in a DIM statement, BASIC09 assumes the variable is real.

The Byte Data Type

Byte variables hold unsigned eight-bit data (integers in the range 0 through 255). Using byte values in computations, BASIC09 converts the byte values to 16-bit integer values. If you store an integer value that is too large for the byte range, BASIC09 stores only the least-significant eight bits (a value of 255 or less), and does not return an error.

The Integer Data Type

Integer variables require two bytes (16 bits) of storage. They can fall in the range -32768 to 32767. If a calculation involves both integer values and real values, BASIC09 presents the result of the calculation as a real number.

You can also use hexadecimal values in integer data. To do so, precede the value with the dollar sign ($). For instance, to represent the decimal value 199 as hexadecimal, type `$C7`. The hexadecimal value range is $0000 through $FFFF.

If you give an integer variable a value that is outside the integer range (greater than 32767 or less than -32768), BASIC09 does not produce an error. Instead it *wraps around* the value range. For instance, the calculation 32767 + 1 produces a result of -32768.

This means that numeric comparisons made on values in the range 32768 through 65535 deal with negative numbers. You should limit such comparisons to tests for equality or non-equality. Functions such as LAND, LNOT, LOR, and LXOR use integer values but produce results on a non-numeric, bit-by-bit, basis.

Division of an integer by another integer yields an integer. BASIC09 discards any remainder.

The Real Data Type

If you do not assign a data type to a variable, BASIC09 assumes the variable is real. However, programs are easier to understand if you define all variable types.

BASIC09 stores as real values any constants that have decimal points. If a constant does not have a decimal point, BASIC09 stores it as an integer.

BASIC09 requires five consecutive memory bytes to store real numbers. The first byte is the exponent, in binary two's complement. The next four bytes are the binary sign and magnitude of the mantissa. The mantissa is in the first 31 bits; the sign of the mantissa is in the last (least-significant) bit of the last byte. The following illustration shows the memory storage of a real number:

Internal Representation of Real Numbers

	exponent	mantissa			S
byte:	0	1	2	3	4

The exponent covers the range $2.938735877 \times 10^{-39}$ (2^{-128}) through $1.701411835 \times 10^{38}$ (2^{127}) as powers of 2. Operations that result in values out of the representation range cause an overflow or underflow error. You can handle such errors using the ON ERROR command.

The mantissa covers the range 0.5 through .9999999995 in steps of 2^{-31}. This means that real numbers can represent values .0000000005 apart. BASIC09 rounds operation values that fall between these points to the nearest point.

Because floating point arithmetic is inherently inexact, a sequence of operations can produce a cumulative error. Proper rounding, as implemented in BASIC09, reduces the effect of this problem, but cannot eliminate it. When using real quantities in comparisons, be sure your computations can produce the exact value you desire.

String Variables

A string is a variable-length sequence of ASCII values. The length can vary from 0, a *null* string, to the capacity of the memory available to BASIC09.

You can define a string variable either explicitly, using the DIM statement, or implicitly by appending the dollar sign ($) to the variable identifier (variable name). For example, title$ implicitly identifies a string variable.

Unless you specify otherwise, BASIC09 assigns a maximum string length of 32 characters. Using the DIM statement, you can specify a maximum length either less than or greater than 32. To conserve memory, use DIM to assign only the maximum length you need for any string variable.

The beginning of a string is always Character 1. The BASE statement, which sets numeric variable base numbers as either 0 or 1, does not affect string variables.

If an operation results in a string too long to fit in the assigned maximum storage space, the system truncates the string on the right. It does **not** produce an error.

String storage is fixed at the dimensioned length. The sequence of actual string byte values is terminated by the value of zero, or by the maximum length allotted to the string. Any unused storage after the zero byte allows the stored string to expand and contract within its assigned length.

The following example shows the internal storage of a variable dimensioned as `string[6]` and assigned the value "SAM". Note that Byte 4 contains the string terminator 00. The string does not use bytes following 00.

	S	A	M	00		
byte:	1	2	3	4	5	6

If you assign the value "ROBERT" to the variable, BASIC09 does not need to terminate the string with 00 because the string is full:

	R	O	B	E	R	T
byte:	1	2	3	4	5	6

The way BASIC09 handles string storage is important when you write programs. If you do not specify a length for strings you define, the system uses the default length 32. As you can see, this wastes computer memory if you store strings of only four or five characters.

The Boolean Type

A Boolean operation always returns either the character string "TRUE" or "FALSE". You cannot use the Boolean data type for numeric computation—only for comparison logic.

Do not confuse the Boolean operations AND, OR, XOR, and NOT (which operate on the Boolean values TRUE and FALSE) with the logical functions LAND, LOR, LXOR, and LNOT (which use integer values to produce numeric results on a bit-by-bit basis). An attempt to store a non-Boolean value in a Boolean variable, causes an error.

Automatic Type Conversion

When an operation mixes numeric data types (byte, integer, or real values), BASIC09 automatically and temporarily converts the values to the type necessary to retain accuracy. This conversion lets you use numeric quantities of mixed types in most calculations.

The system returns a type-mismatch error when an expression includes types that cannot legally mix. These errors are reported by the second compiler pass, which occurs automatically when you exit the edit mode.

Because type conversion takes additional execution time, you can speed calculations by using values of a single type.

Constants

Constants are values in a program that do not change. They can use any of the five data types. The following are examples of constants in a procedure:

```
HOME$="Fort Worth"
VALUE$="$25,000
VALUE=25
PAYMENT=99.99
ANSWER="TRUE"
MEMORY=$0CFF
PRINT "The End"
```

Numeric constants are either integers or real numbers. If a numeric constant includes a decimal point or uses the "E format" exponential form, it causes BASIC09 to store the number in the real format, even if it could store the number in integer or byte format.

You can use this feature to *force* a real format. For instance, to make the number 12 a real number, type it as 12.0. You might want to force real values in this way when all other values in an expression are real so that BASIC09 does not have to do a time-consuming type conversion at run time.

BASIC09 also stores as real numbers any numbers that do not have decimal points but that are too large to store as integers. Here are some examples of legal real constants:

1.0	9.8433218	-.01
-999.000099	100000000	5644.34532
1.95E+12	-99999.9E-33	

BASIC09 treats numbers that do not have a decimal point and are in the range -32768 through +32767 as integers. You must always precede hexadecimal numbers with a dollar sign.

Following are examples of legal integer constants:

12	-3000	55
$20	$FF	$09
0	-12	-32768

String Constants

A string constant consists of a sequence of characters enclosed in double quotation marks, such as:

```
"The End"
```

To place a string constant in a string type variable, use the equal symbol in this manner:

```
TITLE$ = "Masters Of Magic"
```

To include double quotation marks within a string, use two sets of double quotation marks, like this:

```
"An ""older man"" is wiser."
```

A string can contain characters that have ASCII values in the range 0 through 255.

Variables

In BASIC09, a variable is *local* to the procedure in which it is defined. A variable defined in one procedure has no meaning in another procedure unless you use the RUN and PARAM statements to pass the variable when you call the other procedure.

The local nature of variables lets you use the same variable name in more than one procedure and, unless you specify otherwise, have the variables operate independently of each other.

You can assign variables using either the LET statement with the assign symbol (=), or by using the assign symbol alone. For instance, both the following command lines are legal:

```
LET PAYMENT=44.50
PAYMENT=44.50
```

When you call a procedure, BASIC09 allocates storage for the procedure's variables. It is not possible to force a variable to occupy an absolute address in memory. When you exit a procedure, the system returns the storage allotted for variables, and you lose the stored values.

If you write a procedure to call itself (a *recursive* procedure), the call creates separate storage space for variables.

> **Note:** Unlike other BASICS, BASIC09 does not automatically initialize variables by setting them to zero. When you execute a procedure, all variables, arrays, and structures have random values. Your procedure must initialize the variables you specify to the values you require.

Passing Variables

When one procedure passes variable values to another procedure, BASIC09 refers to the passed variables as *parameters*. You can pass variables either by *reference* or by *value*.

BASIC09 does not protect variables passed by reference. Therefore, the called procedure can change the values and return the new values. BASIC09 **does** protect variables passed by value, so, the called program cannot change them.

To pass a parameter by reference, enclose the name of the variable in parentheses as part of the RUN statement in this manner:

`RUN RANDOM(10)`	passes the value 10 to a procedure called Random

The system evaluates the storage address of each passed variable, and sends the variable to the called procedure. The called procedure associates the storage addresses with the names in its local PARAM statement. It then uses the storage area as though it had created it locally. This means it can change the value of the parameter before returning it to the calling procedure.

To pass parameters by value, write the value to be passed as an expression. BASIC09 evaluates the expression at the time of the call. To use a variable in an expression without changing its value, use null constants, such as 0 for a number or "" for a string, in this manner:

`RUN ADDCOLUMN(x+0)`	passes the value of x by value
`RUN TRANSLATE(w$+"")`	passes the contents of w$ by value

To pass parameters by value, BASIC09 creates a temporary variable. It places the result of the expression in the temporary variable and sends the address to the called procedure. This means that the value given to the called procedure is a *copy* of the result of the expression, and the called procedure cannot change the original value.

The results of expressions containing numeric constants are either integer or real values; there are no byte constants. To send byte-type variables to a procedure, pass the values by reference. Therefore, if a RUN statement evaluates an integer as a parameter and sends it to a byte-type variable, the byte variable uses only the high-order byte of the two-byte integer.

Arrays

An *array* is a group of related data values stored consecutively in memory. The system knows the entire group by a variable name. Each data value is an *element*. You use a *subscript* to refer to any element of the array. For example, an array named Graf might contain five elements referred to as:

```
GRAF(1)   GRAF(2)   GRAF(3)   GRAF(4)   GRAF(5)
```

You can use each of these elements to store a different value, such as:

```
GRAF(1) = 25
GRAF(2) = 47
GRAF(3) = 39
GRAF(4) = 18
GRAF(5) = 50
```

Note: Normally, array elements start with 1 in BASIC09. However, you can use the BASE command to cause array elements to begin at 0.

The previous example illustrates a single-dimensioned array. The elements are arranged in one row and only one subscript is used for each element.

The following procedure lets you type values for a GRAF array, and displays the results in a simple graph.

```
PROCEDURE GRAF
□DIM GRAF(5):REAL
□SHELL "DISPLAY ØC"
□FOR T=1 TO 5
□PRINT "Value for Item #"; T; "□";
□INPUT GRAF(t)
□NEXT T
□PRINT
□PRINT
□PRINT "This is how your graph stacks up..."
□PRINT
□FOR T=1 TO 5
□PRINT "Item #"; T; "□";
□FOR U=1 TO GRAF(T)
□PRINT CHR$(79);
□NEXT U
□PRINT
□NEXT T
□PRINT
□END
```

This program uses a single dimension array—in effect, a list.

You can also create arrays with more than one dimension — more than one element for each row. You might use a two-dimensioned array in a program to store names and addresses. Instead of creating separate arrays for the name, address, and zip code, you could set up one array with two dimensions.

The following program, used to enter the names of a company's employees, shows how this might be done. See the second line for the DIM syntax. When you run the procedure, it asks you for a name, address, and zip code for each of 10 employees. After you type the information for all the entries, the procedure displays the information on the screen.

```
PROCEDURE Names
□DIM NAME(10,3):STRING
□SHELL "DISPLAY 0C"
□BASE 0
□FOR T=0 TO 9
□PRINT "Type Employee Name No."; T; ": ";
□INPUT NAME(T,0)
□PRINT "Type Employee Address No."; T; ": ";
□INPUT NAME(T,1)
□PRINT "Type Employee Zip Code No."; T; ": ";
□INPUT NAME(T,2)
□NEXT T
□SHELL "DISPLAY 0C"
□PRINT "And the names are..."
□PRINT
□FOR T=0 TO 9
□PRINT NAME(T,0); "□"; NAME(T,1); "□"; NAME(T,2)
□NEXT T
□END
```

The DIM statement reserves space in memory for a string array named Name, with two dimensions. As you enter data, the Name field is stored in `Name(0,0)`, `Name(1,0)`, `Name(2,0)`, and so on. The Address field is stored in `Name(0,1)`, `Name(1,1)`, `Name(2,1)`, and so on. The Zip field is stored in `Name(0,2)`, `Name(1,2)`, `Name(2,2)`, and so on. This continues until you fill the *grid*, 10 entries with three items each.

You can also create arrays with three dimensions. The following program adds one more dimension that keeps track of each employee's company. It dimensions `Name$` as `Name$(2,10,3)`. The first dimension contains either 0 or 1 to indicate to which company the employee belongs.

```
PROCEDURE names2
□DIM NAME$(2,10,3):STRING
□SHELL "DISPLAY 0C"
□BASE 0
□FOR X=0 TO 1
□PRINT
□PRINT
□FOR T=0 TO 9
□PRINT
□IF X=0 THEN
□PRINT "Type a Wiggleworth Company employee
name..."
□ELSE
□PRINT "Type a Putforth Company employee name..."
□ENDIF
□PRINT "Type Name No."; T; ": ";
□INPUT NAME$(X,T,0)
□PRINT "Type Address No."; T; ": ";
□INPUT NAME$(X,T,1)
□PRINT "Type Zip Code No."; T; ": ";
□INPUT NAME$(X,T,2)
□NEXT T
□NEXT X
□SHELL "DISPLAY 0C"
□PRINT "The Wiggleworth employees are..."
□PRINT
□X=0
□FOR T=0 TO 9
□PRINT NAME$(X,T,0); "□"; NAME$(X,T,1); "□";
NAME$(X,T,2)
□NEXT T
□PRINT
□PRINT "The Putforth Company employees are..."
□PRINT
□X=1
□FOR T=0 TO 9
□PRINT NAME$(X,T,0); "□"; NAME$(X,T,1); "□";
NAME$(X,T,2)
□NEXT T
□END
```

The easiest way to understand three dimensional arrays is to consider the first dimension as a *page*. In other words, if the first dimension in the string is 0, the employee is on the Wiggleworth Company's page. If the first dimension in the string is 1, the employee is on the Putforth Company's page.

Complex Data Types

In addition to the five standard data types, you can create your own data types. Using the TYPE command, you can define a new data type as a *vector* (a single-dimensioned array) of any previously defined type.

For example, in the previous program, the Name variable can contain only one type of data, the string type. However, using the TYPE command you can create a variable that accepts several data types.

Suppose you create an employee list procedure that uses the following variables, of the following size and types:

Name	Length	Contents	Type
Name	25	employee name	string
Street	20	street address	string
City	10	city of address	string
Zip	—	address zip code	integer
Sex	—	false = male, true = female	Boolean
Age	—	employee age	byte

You can combine all these variables into one complex data type. To do so, dimension the variables within a TYPE command line, like this:

```
TYPE EMPLOYEE=NAME:STRING[25]; STREET:STRING[20];
CITY:STRING[10]; ZIP:REAL; SEX:BOOLEAN; AGE:BYTE
```

This creates a new BASIC09 type, called Employee. Employee requires its variables to have six fields of the name, size, and type shown in the previous chart.

Once you create the new data type, you can define variables to use it. For instance, the following program line defines Worker as type employee, with 10 elements in the array:

```
□DIM WORKER(10):EMPLOYEE
```

To put the employee data type to work, collect your data with INPUT commands. Then, store the data into the new Worker array. The following program demonstrates how you might do this:

```
PROCEDURE worker
□REM           Dimension variables for input
□DIM NM:STRING[25]
□DIM ST:STRING[20]
□DIM CTY:STRING[10]
□DIM ZP:REAL
□DIM SX:BOOLEAN
□DIM AG:BYTE
□REM          Create new type and array using new
type
□TYPE EMPLOYEE=NAME:STRING[25]; STREET:STRING[20];
CITY:STRING[10 ]; ZIP:REAL; SEX:BOOLEAN; AGE:BYTE
□DIM WORKER(10):EMPLOYEE
□REM
□FOR T=1 TO 10
□INPUT "Name:□",NM
□INPUT "Street:□",ST
□INPUT "City:□",CTY
□INPUT "Zip:□",ZP
□INPUT "Sex:□",SX
□INPUT "Age:□",AG
□REM          Store input in the Worker array using
field names
□WORKER(T).NAME=NM
□WORKER(T).STREET=ST
□WORKER(T).CITY=CTY
□WORKER(T).ZIP=ZP
□WORKER(T).SEX=SX
□WORKER(T).AGE=AG
□PRINT
□PRINT "* * * * * * * * * * * * * * * * * * * *"
□PRINT
□NEXT T
□SHELL "DISPLAY (0C)"
□PRINT "The names in your files now are..."
□PRINT
□FOR T=1 TO 10
□PRINT WORKER(T).NAME
□PRINT WORKER(T).STREET
□PRINT WORKER(T).CITY
```

```
□PRINT WORKER(T).ZIP
□IF WORKER(T).SEX=TRUE
□THEN PRINT "Female"
□ELSE
□PRINT "Male"
□ENDIF
□PRINT WORKER(T).AGE
□PRINT
□PRINT "* * * * * * * * * * * * * * * * * * * * *"
□PRINT
□NEXT T
```

Note that the Sex field is defined as Boolean. This means that you can respond only in two ways, TRUE or FALSE. The method of input requires only one byte of storage. To use this data you need to handle it so TRUE and FALSE indicate female and male.

Complex data types can contain more than one field. Each field can be of any data type. You reference the fields of a complex data type by typing the variable name, its array index, a period (.), and the field name. The following lines, from the Worker procedure, show how BASIC09 stores the data from the input lines into the Worker variable:

```
WORKER(T).NAME=NM
WORKER(T).STREET=ST
WORKER(T).CITY=CTY
WORKER(T).ZIP=ZP
WORKER(T).SEX=SX
WORKER(T).AGE=AG
```

These lines store the values in the variables NM, ST, CTY, ZP, SX, and AG into the NAME, STREET, CITY, ZIP, SEX, and AGE fields of the Worker variable. This operation is within a FOR/NEXT loop that uses T as a counter. In the procedure, T can refer to a value in the range 1 to 10.

The procedure uses the same type of operation to extract the data from the complex data type variable:

```
PRINT WORKER(T).NAME
PRINT WORKER(T).STREET
PRINT WORKER(T).CITY
PRINT WORKER(T).ZIP
IF WORKER(T).SEX=TRUE THEN PRINT "Female"
ELSE PRINT "Male"
ENDIF
PRINT WORKER(T).AGE
```

Using the same methods, you can create complex data types that combine other complex data types and standard data types.

The elements of a complex structure can be copied to another similar structure. Using a single assignment operator, you can write an entire structure to, or read an entire structure from, mass storage as a single entity. For example:

```
PUT #2, WORKER(T)
```

Because the system defines the elements of complex-type storage during compilation, it need not do so during *runtime*. This means that BASIC09 can reference complex structure faster than it can reference arrays.

Chapter 7

Expressions, Operators, and Functions

Manipulating Data

BASIC09 uses *expressions* to manipulate data. (Expressions are pieces of data connected by operators.)

An *operator* is a symbol or a word that signifies some action to be performed on the specified data. Each data item is a *value*.

Expressions

When an expression is evaluated, the result is a value of some data type (real, integer, string, byte, or Boolean).

An expression might look like this:

First Value	First Operator	Second Value	Second Operator	Result
6	+	5	=	11

or like this:

First Value	First Operator	Second Value	Second Operator	Result
"Seaside"	+	"Villa"	=	Seaside Villa

When BASIC09 evaluates an expression, it copies each value onto an *expression stack*. Functions and operators take their input values from this stack and return their results to it. Many expressions result in assignments, as do the examples shown. The BASIC09 makes the resulting assignment only after it computes the entire expression. This lets you use the variable that is being modified as one of the values in the expression, such as in this example:

```
X = X + 1
```

The result of an expression is always one of the five BASIC09 data types. However, you can often mix data types within an expression and, in some cases, the result of an expression is of a different data type than any of the values in the expression. Such is the case if you use the *less-than* symbol (<), in this manner:

```
24 < 100
```

The less-than operator compares two integer values. The result of the comparison is Boolean; in this case, the value is TRUE.

Type Conversion

Because BASIC09 performs automatic type conversion of values, you can mix any of the three numeric data types in an expression. When you mix numeric data types, the result is always of the same type as the value having the largest representation, in this order: real < integer < byte.

You can use any numeric type in an expression that produces a real number. If you want an expression to produce a byte or integer type value, the result must be small enough to fit the desired type.

Operators

BASIC09 has operators to deal with all types of data. Each operator, except NOT and negation (unary -), takes two values or *operands*, and performs an operation to produce a result. NOT can accept only one value. The following table lists the operators available and the types of data they accept and produce.

Because the same operators function on the three types of numeric data (byte, integer, and real), these types are referred to by the operand type "numeric."

BASIC09 Expression Operators

Operator	Function	Operand Type	Result Type
−	Negation	numeric	numeric
^ or **	Exponentiation	numeric	numeric
*	Multiplication	numeric	numeric
/	Division	numeric	numeric
+	Addition	numeric	numeric
−	Subtraction	numeric	numeric
NOT	Logical Negation	Boolean	Boolean
AND	Logical AND	Boolean	Boolean
OR	Logical OR	Boolean	Boolean
XOR	Logical Exclusive OR	Boolean	Boolean
+	Concatenation	string	string
=	Equal to	all types	Boolean
<> **or** ><	Not equal to	all types	Boolean
<	Less than	numeric, string†	Boolean
<= **or** =<	Less than or equal	numeric, string†	Boolean
>	Greater than	numeric, string†	Boolean
>= **or** =>	Greater than or equal	numeric, string†	Boolean

† When comparing strings, BASIC09 uses the ASCII values of characters as the basis for comparison. Therefore, 0 < 1, 9 < A, A < B, A < b, b < z, and so on.

Arithmetic Operators

Arithmetic operators perform operations on numeric data. Therefore, both operands in the expression must be numeric. The following table lists the arithmetic operators.

Negation The single dash negates a number's sign: -10 is *negative* 10.

Exponentiation Use a caret (^) or two asterisks (**) to raise a number to a power: 2^3 is 8 (2 x 2 x 2). Similarly, 2**3 is 8.

Multiplication A single asterisk causes multiplication: 2 * 3 is 6.

Division A slash causes division: 6 / 2 is 3.

Addition The plus sign causes addition: 3 + 3 is 6.

Subtraction A dash causes subtraction: 6 - 3 is 2.

Hierarchy of Operators

BASIC09 uses the standard hierarchy of operations when calculating expressions with multiple operators. This means that BASIC09 has an order in which it performs calculations involving more than one operator.

The following BASIC09 operators are listed in order of precedence:

```
NOT    - (negate)
^      **
*      /
+      -
>      <  <>  =  >=  <=
AND
OR     XOR
```

Also, BASIC09:

- Performs operations enclosed in parentheses **before** operations not in parentheses.
- Performs the leftmost operations first when two or more operations are of equal precedence.

You can use parentheses to override this standard precedence. For example:

```
2 + 1 * 3 = 5
```

but

```
( 2 + 1 ) * 3 = 9
```

The following examples show BASIC09 expressions on the left, and the way BASIC09 evaluates them on the right. You can enter the expressions in either form, but the decompiler generates the simpler form, shown on the left.

BASIC09 Representation	Equivalent Form
a=b+c**2/d	a=b+((c**2)/d)
a=b>c AND d>e OR c=e	a=((b>c) AND (d>e)) OR (c=e)
a=(b+c+d)/e	a=((b+c)+d)/e
a=b**c**d/e	a=(b**(c**d))/e
a= -(b)**2	a=(-b)**2
a=b=c	a=(b=c)

Relational Operators

Relational operators make logical comparisons of any type of data and return a result of either TRUE or FALSE. An explanation of the relational operators follows. All relational operators have equal precedence.

=	Equal. Returns TRUE if both operands are equal, or FALSE if they are not equal.
<	Less than: Returns TRUE if the first operand is less than the second, or FALSE if is not.
>	Greater than: Returns TRUE if the first operand is greater than the second, or FALSE if it is not.
<> **or** ><	Unequal: Returns TRUE if the operands are not equal or FALSE if they are.
<= **or** =<	Less than or equal to: Returns TRUE if the first operand is less than or equal to the second operand. Otherwise, the operation returns FALSE.
>= **or** =>	Greater than or equal to: Returns TRUE if the first operand is greater than or equal to the second. Otherwise, the operation returns FALSE.

You normally use relational operators in IF/THEN statements. For example, if your procedure has two numeric variables, Payments and Income, you might include command lines like this:

```
IF PAYMENTS > INCOME THEN
  PRINT "You're Broke!"
ENDIF
```

When you combine arithmetic and relational operators in the same expression, BASIC09 evaluates the arithmetic operations first. For example:

```
IF X*Y/2 <= 14 THEN
  PRINT "Average Score is "; X*Y/2
ENDIF
```

BASIC09 performs the arithmetic operation x*y/2, then compares the result with the value 14.

When you use relational operators with strings, BASIC09 compares the strings character by character. When it finds two characters that do not match, it checks to see which character has the lower ASCII code value. The string containing the character with the lower value comes first.

Consider this example:

```
PRINT "hunt" > "hung"
```

BASIC09 compares each character in each string. Because the first three characters are the same, the result of the operation is based on the comparison of t and g. Because t (ASCII value = 116) is "greater than" g (ASCII value = 103), the command prints TRUE.

String Operators

The *string operator* is the plus sign (+). This symbol appends one string to another. All operands must be strings, and the resulting value is one string. Examine, for example, the following line, which appends three strings:

```
PRINT "My friends are " + "Jack and " + "Jill."
```

It prints: `My friends are Jack and Jill.`

Logical Operators

The logical, or Boolean, operators make logical comparisons of Boolean values. The following table describes the results yielded by each logical operator given the specified TRUE/FALSE values:

Operator	Meaning of Operation	First Operand	Second Operand	Result
NOT	The result is the opposite of	TRUE		FALSE
	the operand.	FALSE		TRUE
AND	When both values are TRUE,	TRUE	TRUE	TRUE
	the result is TRUE.	TRUE	FALSE	FALSE
	Otherwise, the result is	FALSE	TRUE	FALSE
	FALSE.	FALSE	FALSE	FALSE
OR	When both values are	TRUE	TRUE	TRUE
	FALSE, the result is FALSE.	TRUE	FALSE	TRUE
	Otherwise, the result is	FALSE	TRUE	TRUE
	TRUE.	FALSE	FALSE	FALSE
XOR	When only one of the values	TRUE	TRUE	FALSE
	is TRUE, the result is	TRUE	FALSE	TRUE
	TRUE. Otherwise the result	FALSE	TRUE	TRUE
	is FALSE.	FALSE	FALSE	FALSE

Use logical operators in IF/THEN statements such as:

```
IF PAYMENTS < INCOME AND INCOME+SAVINGS >
PAYMENTS THEN
  PRINT "You'll have to use your savings to get
out of this mess."
ENDIF
```

Functions

Functions are operation sequences the system performs on data. In a statement, BASIC09 performs functions first. Chapter 11, “Command Reference,” describes the following functions.

Functions returning results of type real

SIN	Calculates the trigonometric sine of a number.
COS	Calculates the trigonometric cosine of a number.
TAN	Calculates the trigonometric tangent of a number.
ASN	Calculates the trigonometric arcsine of a number.
ACS	Calculates the trigonometric arccosine of a number.
ATN	Calculates the trigonometric arctangent of a number.
LOG	Calculates the natural logarithm (base e) of a number.
LOG10	Calculates the logarithm (base 10) of a number.
EXP	Calculates e (2.71828183) raised to the specified positive power.
FLOAT	Converts byte or integer type numbers to real numbers.
INT	Calculates the largest whole number less than or equal to the specified number.
PI	Represents the constant 3.14159265.
SQR	Calculates the square root of a positive number.
SQRT	Calculates the square root of a positive number. Its function is identical to SQR.
RND	Returns a random number.

Functions returning results of any numeric type

The resulting type depends on the input type.

ABS	Calculates the absolute value of a number.
SGN	Returns a value to indicate the sign of the specified number (-1 if the number is less than 0, 0 if the number is 0, or 1 if the number is greater than 0).
SQ	Calculates the square of a number.
VAL	Converts a string to a numeric value.

Functions returning results of type integer or type byte

FIX	Rounds a real number and converts it to an integer.
MOD	Calculates the modulus (remainder) of two numbers.
ADDR	Returns the absolute memory address of a variable, an array, or a structure.
SIZE	Returns (in bytes) the storage size of a variable, an array, or a structure.
ERR	Returns the error code of the most recent error.
PEEK	Returns the byte value at a specified memory address.
POS	Returns the current character position of the print buffer.
ASC	Returns the numeric value (ASCII code) of a string character.
LEN	Returns the length of a string.
SUBSTR	Returns the starting position of the specified substring within a string, or returns 0 if it cannot find the substring.

Functions performing bit-by-bit logical operations on integer or byte data and returning integer results. Do not confuse these functions with Boolean type operators.

LAND	Calculates the logical AND of two values.
LOR	Calculates the logical OR of two values.
LXOR	Calculates the logical EXCLUSIVE OR of two values.
LNOT	Calculates the logical NOT of a value.

Functions returning a result of type string

CHR$	Returns the character having a specified ASCII value.
DATE$	Returns the system's current date and time.
LEFT$	Returns the specified number of characters beginning at the leftmost character of the specified string.
RIGHT$	Returns the specified number of characters beginning at the rightmost character of the specified string and counting backward.
MID$	Returns the specified number of characters starting at the specified position in a string.
STR$	Converts numeric type data to string type.
TRIM$	Removes trailing spaces from the specified string.

Functions returning Boolean values

TRUE	Always returns TRUE.
FALSE	Always returns FALSE.
EOF	Tests for the end of a disk file. Returns TRUE when the end of the file occurs.

Chapter 8

Disk Files

When you tell OS-9 or BASIC09 to store (save) data on a disk, it stores the data in a *logical* block called a *file*. The term logical means that, although the system might store portions of a file's data in several different disk locations, it keeps track of every location and treats the scattered data as though it occupied a single block. It does this automatically and you never need to worry about how the data is stored. File data can be binary data, textual data (ASCII characters), or any other useful information.

Because OS-9 handles all hardware input/output devices (disk drives, printers, terminals, and so on) in the same manner, you can send data to any of these devices in the same way. This means you can send the same information to several devices by changing the path the data follows. For example, you can test a procedure that communicates with a terminal by transferring data to and from a disk drive.

BASIC09 normally works with two types of files—sequential files and random access files. The following chart shows file-access options, their purposes, and the keywords with which to use them:

Types of Access for Files

Access Type	Function	Use with
DIR	Opens a directory file for reading. Use only with READ.	OPEN
EXEC	Specifies that the file to open or create is in the execution directory, rather than the data directory.	OPEN CREATE
READ	Lets you read data from the specified file or device.	OPEN CREATE
WRITE	Lets you write data to the specified file or device.	OPEN CREATE
UPDATE	Lets you read data from and write data to the specified file or device.	OPEN CREATE

Sequential Files

Sequential files send or receive (WRITE or READ) textual data in order, the second item following the first, and so on. You can access sequential data only in the same order as you originally stored it. To read from or write to a particular section of a file, you must first read through all the preceding data in the file, starting from the beginning.

BASIC09 stores sequential file data as ASCII characters. Each block of data is separated by a *delimiter* consisting of a carriage-return character (ASCII Character 13). Because BASIC09 uses this delimiter to determine the end of a *record*, sequential files can contain records of varying length.

Use the WRITE and READ commands to store and retrieve data in sequential files. A WRITE command causes BASIC09 to transfer specified data to a specified file, ending the data with a carriage return. A READ command causes BASIC09 to load from the specified file the next block of data, stopping when it reaches a carriage return.

Sequential File Creation, Storage, and Retrieval

BASIC09 uses the CREATE command to establish both sequential and random access files. A CREATE statement contains:

- The keyword CREATE.
- A path number variable in which BASIC09 stores the number of the path it opens to the new file.
- A comma, followed by the name of the file to create.
- An optional colon, followed by the access mode. If you do not specify an access mode, BASIC09 automatically opens the created file in the UPDATE mode.

The following procedure shows how to create a file and write data into it:

```
PROCEDURE makefile
□DIM PATH:BYTE                   (* establishes a variable
□REM                                for the path number to the file
□CREATE #PATH,"test":WRITE       (* creates the file TEST
□WRITE #PATH,"This is a test"    (* writes data to the file
□WRITE #PATH,"of sequential files."(* writes another line of data
□CLOSE #PATH                     (* closes the path to the file
□SHELL "LIST TEST"               (* displays the file contents
□END
```

The first line of the procedure dimensions a variable (Path) to hold the number of the path that CREATE opens. This variable should be of byte or integer type.

When you establish a new file with CREATE, you automatically open a path to the file. You do not need to use the OPEN command.

The preceding procedure writes two lines into a file named Test. It then closes the path and uses the OS-9 LIST command to display the contents of the newly created file. You see that the data is successfully stored on disk.

The next procedure shows how to reopen an existing file for sequential access, read the contents of the file, and append data to the end of the file.

The only way to move the *file pointer* to the end of a sequential file is to read all the data already in the file. Once the pointer is at the end of the file, you can add data.

```
PROCEDURE append
□DIM PATH:BYTE              (* dimension variable to hold the number of the
□REM                           path to the opened file.
□OPEN #PATH,"test":UPDATE   (* open file for reading and writing.
□READ #PATH,line$           (* read the first element of the file.
□READ #PATH,line$           (* read the next (the last) element.
□WRITE #PATH,"This is a test" (* write one new line to the file.
□WRITE #PATH,"of appending to a sequential file." (* write another.
□CLOSE #PATH                (* close the path.
□SHELL "LIST TEST"          (* display the file with the new lines.
□END
```

Because the Test file already exists, this procedure uses OPEN to establish a path to the file. It uses the UPDATE mode of file access because it needs to both read from and write to the file.

The two READ statements read the file's contents and, as a result, move the file pointer to the end of the file. The WRITE statements then append two new lines. After closing the path, the procedure calls on the OS-9 LIST command to display the contents of the file, with its appended lines.

Changing Data in a Sequential File

You can also change data anywhere in a sequential file. However, if your changes are longer than the original data, the operation destroys part of the file. To change data in a sequential file, read the data preceding what you want to change, and write the new data to the file in this manner:

```
PROCEDURE replace
□DIM PATH:BYTE
□OPEN #PATH,"test":UPDATE
□READ #PATH,line$
□READ #PATH,line$
□WRITE #PATH,"Let's put new"   (* write over existing 3rd and
□WRITE #PATH,"words into the old sequential file.'' (* 4th lines.
□CLOSE #PATH
□SHELL "LIST TEST"
□END
```

Notice that the total amount of data in the two new lines is exactly the same as in the two old lines. You can replace an existing line with fewer characters by *padding* the new data with spaces. However, if you try to replace existing lines with longer lines, the new lines write over and destroy other data in the file.

INPUT and Sequential Files

Although you can also use the INPUT command with sequential files, doing so might put unwanted data into them. When a procedure encounters INPUT, it suspends execution and sends a question mark (?) to the screen. This feature makes INPUT both an input and output statement. Therefore, if you open a file using the UPDATE mode, INPUT writes its prompts to the file, destroying data. If you specify text to be displayed with the INPUT command, INPUT writes this text to the file also.

Random Access Files

Random access files store data in fixed- or equal-length blocks. Because each record in a specific file is the same size, you can easily calculate the position of a record.

For instance, suppose you have a file with a record length of 50-bytes (or characters). To access Record 10, multiply the record number (10) by the record length (50) and move the file pointer to the calculated position (500).

A random access file sends and receives data (using PUT and GET) in a binary form, exactly as BASIC09 stores it internally. This feature minimizes the time involved in converting the data to and from ASCII representation, as well as reducing the file space required to store numeric data. You position the random access file pointer using SEEK. Compared to sequential file access, random file access using GET and PUT is very fast.

Using random access commands, you can store and retrieve individual bytes, strings of bytes, individual elements of arrays or total arrays with one PUT or GET command. When you GET a structure, you recover the number of bytes associated with that type of structure.

This means when you GET one element of byte type data, you read one byte. When you GET one element of real type data, you read five bytes. If you GET an array, you read all the elements of the array. This potential for reading entire arrays at once can greatly speed disk access.

As well as moving the file pointer to the beginning of individual records, you can also move it to any position within a record and begin reading or writing one or more bytes from that point.

Creating Random Access Files

You create and open random access files in the same way you create and open sequential files. The only differences are in the commands you use to store and retrieve the data and in the manner you keep track of where elements, or records, of a file begin and end.

Before you can write data to a random access file, you must either CREATE it or open it in the WRITE or UPDATE mode. Once you have a path open to an existing file, use PUT to write data into the file. If you open the file in the READ or UPDATE mode, you can then use the GET command to retrieve data from the file.

The PUT command can use only one parameter, the name of the data element to store. The parameter can be a string, a variable, an array, or a complex data structure.

Before storing data, you must devise a method to store it in blocks of equal size. Knowing the unit size lets you later retrieve the data in its original form. The following procedure shows one way to do this:

```
PROCEDURE putget
□REM This procedure creates a file named Test1, reads 10 data lines,
□REM PUTs them into the file, then closes the file. Next it
□REM opens the file in the READ mode, GETS stored lines and lists
□REM them on the display screen.

□DIM LENGTH:BYTE
□DIM NULL:STRING[25]
□DIM LINE:STRING[25]
□DIM PATH:BYTE
□LENGTH=25
□NULL=""
□BASE 0
□ON ERROR GOTO 10
□DELETE "test1"            (* if the file exists, delete it.
10□ON ERROR

□CREATE #PATH,"test1":WRITE (* create a file named test1.
□FOR T=0 TO 9
□SEEK #PATH,LENGTH*T        (* find beginning of each file.
□READ LINE$                 (* read a line of data.
□PUT #PATH,LINE$            (* store the line in the file.
□NEXT T
```

```
□CLOSE #PATH              (* close the file.

□OPEN #PATH,"test1":READ  (* open the file for reading.
□FOR T=Ø TO 9
□SEEK #PATH,LENGTH*T      (* find the beginning of each file.
□GET #PATH,LINE           (* get a line from the file.
□PRINT LINE               (* display the line.
□NEXT T

□CLOSE #PATH              (* close the file.
□END

□DATA "This is test line #1"
□DATA "This is test line #2"
□DATA "This is test line #3"
□DATA "This is test line #4"
□DATA "This is test line #5"
□DATA "This is test line #6"
□DATA "This is test line #7"
□DATA "This is test line #8"
□DATA "This is test line #9"
□DATA "This is test line #10"
```

This procedure creates a file named Test1. The variable named Length stores the length of each line in the file (25 characters). The string variable Null, is a string of 25 space characters. The variable Line contains the data to store in each element (record) in the file. The variable Path stores the path number of the file.

Next, the procedure contains an ON ERROR routine that deletes the file Test1, if it already exists. Without this routine, the procedure produces an error if you execute it more than once.

Next, the routine uses CREATE to open the file Test1. The line `SEEK #PATH, LENGTH*T` sets the file pointer to the proper location to store the next line. Because Length is established as 25, the file lines are stored at 0, 25, 50, 75, and so on.

After the routine initializes storage space, it begins to store data by reading the procedure data lines one at a time, seeking the proper file location, and putting the data into the file. After storing all 10 lines, it closes the file.

The last part of the routine opens the new file, uses the same SEEK routine to position the file pointer, and reads the lines back, one at a time, to confirm that the store routine is successful.

The next short routine shows how you can use a procedure to read any line you select in the file, without reading any preceding lines:

```
PROCEDURE randomread
□DIM LENGTH:BYTE
□DIM LINE:STRING[25]
□DIM SEEKLINE:BYTE
□DIM PATH:BYTE
□LENGTH=25

□OPEN #PATH,"test1":READ             (* open the file for reading.
□LOOP
□INPUT "Line number to display...",SEEKLINE (* type a line to get.
□EXITIF SEEKLINE>10 OR SEEKLINE<1 THEN (* test if record is valid.
□ENDEXIT                             (* exit loop if not.
□SEEK #PATH,(SEEKLINE-1)*LENGTH      (* find the requested record.
□GET #PATH,LINE                      (* read the record.
□PRINT LINE                          (* display the record.
□PRINT
□ENDLOOP
□PRINT "That's all ....... "         (* end session.
□CLOSE #PATH                         (* close path.
□END
```

The procedure asks for the record number of the line to display. When you type the number (1-10) and press ENTER, SEEK moves the file pointer to the beginning of the record you want, GET reads it into the variable Line, and PRINT displays it. The calculation `(SEEKLINE-1)*LENGTH` determines the beginning of the line you want. If you type a number outside the range of lines contained in the file (1-10), the procedure drops down to Line 100 and ends.

By changing this procedure slightly, you can replace any line in the procedure with another line. The altered procedure below demonstrates this:

```
PROCEDURE random_replace
□DIM LENGTH:BYTE
□DIM LINE:STRING[25]
□DIM SEEKLINE:BYTE
□DIM PATH:BYTE
□LENGTH=25

□OPEN #PATH,"test1":UPDATE(* open the file.
□LOOP
□INPUT "Line number to display...",SEEKLINE (* type record to find.
□EXITIF SEEKLINE>10 OR SEEKLINE<1 THEN (* test if valid number.
□ENDEXIT                          (* exit loop if not
□SEEK #PATH,(SEEKLINE-1)*LENGTH (* find the requested record.
□GET #PATH,LINE                   (* get the data.
□PRINT LINE                       (* print the record.
□PRINT
□INPUT "Type new line... ",LINE (* type a new line.
□SEEK #PATH,(SEEKLINE-1)*LENGTH (* find beginning of the record.
□PUT #PATH,LINE                   (* store the new line.

□ENDLOOP                          (* do it all again.

□PRINT "That's all ....... "      (* terminate procedure.
□CLOSE #PATH                      (* close path.
□END
```

This time, the file is opened in the UPDATE mode to allow both reading and writing. You type the line you want to display. A prompt then asks you to type a new line. The procedure exchanges the new line for the original line, and stores it back in the file.

Using Arrays With Random Access Files

BASIC09's random access filing system is even more impressive when used with data structures, such as arrays. Instead of using a loop to store the 10 lines of the Random_replace procedure, you could store them all at once, into one record, using an array. The following procedure illustrates this:

```
PROCEDURE arraywrite
□DIM LENGTH:BYTE
□DIM LINE:STRING[25]
□DIM RECORD(10):STRING[25]
□DIM PATH:BYTE
□LENGTH=25

□ON ERROR GOTO 10
□DELETE "test1"                 (* delete Test1 if it exists.
10□ON ERROR

□CREATE #PATH,"test1":WRITE       (* create Test1.
□BASE 0

□FOR T=0 TO 9
□READ RECORD(T)                   (* Read data lines into RECORD array.
□NEXT T

□SEEK #PATH,0                     (* set pointer to beginning of file.
□PUT #PATH,RECORD                 (* store the entire array into file.
□CLOSE #PATH                      (* close path to file.

□OPEN #PATH,"test1":READ          (* open the file to read.
□FOR T=0 TO 9
□SEEK #PATH,LENGTH*T              (* find each element.
□GET #PATH,LINE                   (* read an element.
□PRINT LINE                       (* print the element.
□NEXT T
□CLOSE #PATH
□END

□DATA "This is test line #1"
□DATA "This is test line #2"
□DATA "This is test line #3"
□DATA "This is test line #4"
□DATA "This is test line #5"
□DATA "This is test line #6"
□DATA "This is test line #7"
□DATA "This is test line #8"
□DATA "This is test line #9"
□DATA "This is test line #10"
```

This procedure reads the 10 lines into an array named Records. Then it places the entire array in the Test1 file, using one PUT statement. To show that the structure of the file is still the same, the original FOR/NEXT loop reads the lines, one at a time, and displays them.

Notice that, because you need to write only one element, you can set the file pointer to 0 (`SEEK #PATH,0`). You can *rewind* a file pointer (set it to 0) at any time in this manner.

You could save additional programming space by also reading the 10 lines back into memory as an array. The following procedure uses a new array, Readlines, to call the file back into memory, and displays the lines.

```
PROCEDURE arrayread
□BASE 0
□DIM READLINES(10):STRING[25]
□DIM PATH:BYTE

□OPEN #PATH,"test1":READ        (* open file.
□GET #PATH,READLINES            (* read file into array.
□CLOSE #PATH

□FOR T=0 TO 9
□PRINT READLINES(T)             (* print each element of the array.
□NEXT T
□END
```

Using Complex Data Structures

In the previous section, you stored and retrieved elements of an array that were all the same size, 25 characters. Often you need to store elements of varying sizes, such as when you create a data base program with several fields in one record.

The following examples create a simple inventory system that requires a random access file having 100 records. Each record includes the name of the item (a 25-byte string), the item's list price and cost (both real numbers), and the quantity on hand (an integer).

First, you use the TYPE command to define a new data type that describes such a record. For example:

```
TYPE INV_ITEM=NAME:STRING[25];LIST,COST:REAL;
QTY:INTEGER
```

Although this statement describes a new record type called Inv_item, it does not assign variable storage for the record. The next step is to create two data structures: an array of 100 records of type Inv_item named Inv_array and a working record named
Work_rec. The following lines do this:

```
DIM INV_ARRAY(100):INV_ITEM
DIM WORK_REC:INV_ITEM
```

To determine the number of bytes assigned for each type, you can use BASIC09's SIZE command. SIZE returns the number of bytes assigned to any variable, array, or complex data structure. For example, the command line `SIZE(WORKI_REC)` returns the number 37. The command `SIZE(INV_ARRAY)` returns the number `3700`.

You can use SIZE with SEEK to position a file pointer to a specific record's address.

The following procedure creates a file called Inventory and immediately initializes it with zeroes and null strings. Five INPUT lines then ask you for a record number and the data to store in each field of the record. You can fill any record you choose, from 1 through 100.

When one record is complete, the procedure uses PUT to store the record. Then, it asks you for a new record number. If you wish to quit, enter a number either larger than 100 or smaller than 1.

```
PROCEDURE inventory
□REM Create a data type consisting of a 25-character name field,
□REM a real list price field, a real cost field, and an integer
□REM quantity field.
□TYPE INV_ITEM=NAME:STRING[25]; LIST,COST:REAL; QTY:INTEGER

□DIM INV_ARRAY(100):INV_ITEM (* dimension an array using new type.
```

```
□DIM WORK_REC:INV_ITEM
□REM           (* dimension a working variable of the new type.
□DIM PATH:BYTE

□ON ERROR GOTO 10
□DELETE "inventory"
10□ON ERROR

□CREATE #PATH,"inventory"    (* create a file named Inventory.
□WORK_REC.NAME=" "           (* set all data elements to null or 0.
□WORK_REC.LIST=0
□WORK_REC.COST=0
□WORK_REC.QTY=0
□FOR N=1 TO 100
□PUT #PATH,WORK_REC
□NEXT N

□LOOP
□INPUT "Record number? ",RNUM (* enter number of record to write.
□IF RNUM<1 OR RNUM>100 THEN   (* check if number is valid.
□PRINT
□PRINT "End of Session"       (* if not, end session.
□PRINT
□CLOSE #PATH
□END
□ENDIF
□INPUT "Item name? ",WORK_REC.NAME  (* type data for record.
□INPUT "List price? ",WORK_REC.LIST
□INPUT "Cost price? ",WORK_REC.COST
□INPUT "Quantity? ",WORK_REC.QTY
□SEEK #PATH,(RNUM-1)*SIZE(WORK_REC) (* find record.
□PUT #PATH,WORK_REC                 (* write record to file.
□ENDLOOP
```

Notice that the INPUT statements reference each field separately, but the PUT statement references the record as a whole.

The next procedure lets you read any record in your Inventory file, and displays that record. If you ask for a record you have not yet filled with meaningful data, the display consists of a null string and zeroes.

```
PROCEDURE readinv
□TYPE INV_ITEM=NAME:STRING[25]; LIST,COST:REAL; QTY:INTEGER
□DIM WORK_REC:INV_ITEM
```

```
DIM PATH:BYTE
OPEN #PATH,"INVENTORY":READ
LOOP
INPUT "Record number to display? ",RNUM
IF RNUM<1 OR RNUM>100 THEN
PRINT "End of Session"
PRINT
CLOSE #PATH
END
ENDIF
SEEK #PATH,(RNUM-1)*SIZE(WORK_REC)
GET #PATH,WORK_REC
PRINT "#","Item","List Price","Cost Price","Quantity"
PRINT "-----------------------------------------------------------------
---"
PRINT RNUM,WORK_REC.NAME,WORK_REC.LIST,WORK_REC.COST,WORK_REC.QTY
PRINT
ENDLOOP
END
```

This procedure accesses the file one record at a time. It is not necessary to do so. You can read the entire file into memory at once by dimensioning an inventory array and getting the whole file into it:

```
TYPE INV_ITEM=NAME:STRING[25]; LIST,COST:REAL; QTY:INTEGER
DIM INV_ARRAY(100):INV_ITEM
SEEK #PATH,0    (*rewind the file*)
GET #PATH,INV_ARRAY
```

The examples in this section are simple, yet they illustrate the combined power of BASIC09 complex data structures and the random access file statements. They show that a single GET or PUT statement can move any amount of data, organized in any way you want. Other advantages are of using complex data structures are:

- The procedures are self-documenting. You can see easily what a procedure does because its structures can have descriptive names.
- Execution is extremely fast.
- Procedures are simple and usually require fewer statements to perform I/O functions than other BASICs.

- The procedures are versatile. By creating appropriate data structures, you can read or write almost any kind of data from any file, including files created by other programs or languages.

Displaying Text and Graphics

BASIC09 has three levels of graphics capabilities. The first and third levels can include both graphics designs and text. The second level can display only graphics designs.

ASCII Codes

For low-resolution text screens and high-resolution text and graphic screens, BASIC09 uses ASCII (American Standard Code for Information Interchange) codes to represent the common alphanumeric characters. ASCII is the same code that most small computers use.

A table of the standard codes follows:

Table 9.1
BASIC09 ASCII Codes 0-127
Low- and High-Resolution Screens

Character	Decimal Code	Hexadecimal Code
BREAK	03	03
←	8	O8
→	9	O9
↓	10	OA
CLEAR	12	0C
ENTER	13	0D
Space	32	20
!	33	21
“	34	22
#	35	23
$	36	24
%	37	25
&	38	26
’	39	27
(	40	28
)	41	29
*	42	2A
+	43	2B
,	44	2C
-	45	2D
.	46	2E
/	47	2F
0	48	30

Character	Decimal Code	Hexadecimal Code
1	49	31
2	50	32
3	51	33
4	52	34
5	53	35
6	54	36
7	55	37
8	56	38
9	57	39
:	58	3A
;	59	3B
<	60	3C
=	61	3D
>	62	3E
?	63	3F
@	64	40
A	65	41
B	66	42
C	67	43
D	68	44
E	69	45
F	70	46
G	71	47
H	72	48
I	73	49
J	74	4A
K	75	4B
L	76	4C
M	77	4D
N	78	4E
O	79	4F
P	80	50
Q	81	51
R	82	52
S	83	53
T	84	54
U	85	55
V	86	56
W	87	57
X	88	58
Y	89	59
Z	90	5A
[(SHIFT ↓)	91	5B

Character	Decimal Code	Hexadecimal Code
\ (SHIFT CLEAR)	92	5C
] (SHIFT →)	93	5D
↑	94	5E
→ (SHIFT ↑)	95	5F
^	96	60
a	97	61
b	98	62
c	99	63
d	100	64
e	101	65
f	102	66
g	103	67
h	104	68
i	105	69
j	106	6A
k	107	6B
l	108	6C
m	109	6D
n	110	6E
o	111	6F
p	112	70
q	113	71
r	114	72
s	115	73
t	116	74
u	117	75
v	118	76
w	119	77
x	120	78
y	121	79
z	122	7A
{	123	7B
		124	7C
}	125	7D
■	126	7E
___	127	7F

You can generate the characters in this chart by pressing the appropriate key, or you can generate them from BASIC09 using the CHR$ function.

Low-Resolution Graphic Characters

In addition to alphanumeric characters, low-resolution graphics also offers graphic characters. Generate these characters by pressing ALT at the same time you press a keyboard character. The graphics character codes are in the range 128-255.

Pressing ALT while pressing another key, causes OS-9 to add 128 to the ASCII value of the second key. (For the technically minded, OS-9 sets the high bit of the character code.) Therefore, if you press ALT A, you produce graphics character 193. You can also generate graphics characters from BASIC09 using the CHR$ function, and you can PRINT them in the same manner as other characters.

Low-level graphics characters follow a pattern that repeats every 16 characters. Table 9.2 shows the first set of graphic characters, 128-143. Subsequent characters produce the same series of configurations but display in different colors, as shown in Table 9.3.

Table 9.2
Low-Resolution Graphic Character Set

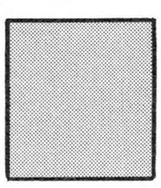
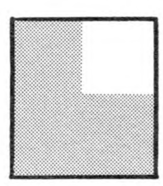
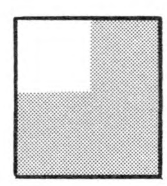
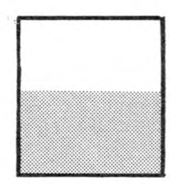
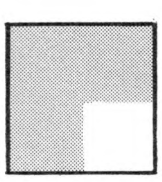
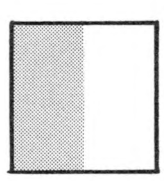
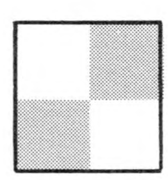
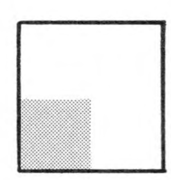
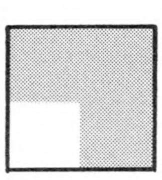
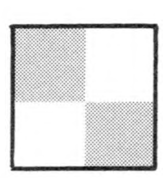
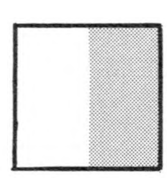
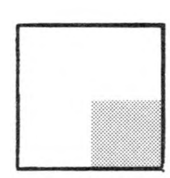
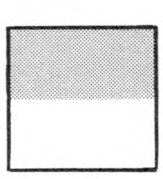
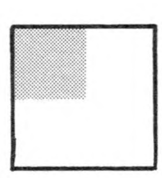
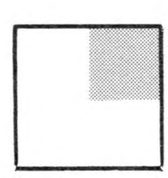
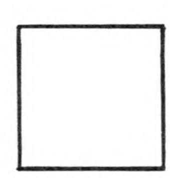
Character	Code	Character	Code	Character	Code	Character	Code
	128		132		136		140
	129		133		137		141
	130		134		138		142
	131		135		139		143

Table 9.3
Low-Resolution Graphics Color Set

ASCII Code	Graphics Block Color
128 - 143	Black and Green
144 - 159	Black and Yellow
160 - 175	Black and Blue
176 - 191	Black and Red
192 - 207	Black and Buff
208 - 223	Black and Light Blue
224 - 239	Black and Cyan
240 - 254	Black and Orange
255	Green

Within each color set, you can easily calculate the number for a particular character. For instance, suppose you want to print a character that has orange upper left and lower right corners. Picture the character divided into four sections, numbered as follows:

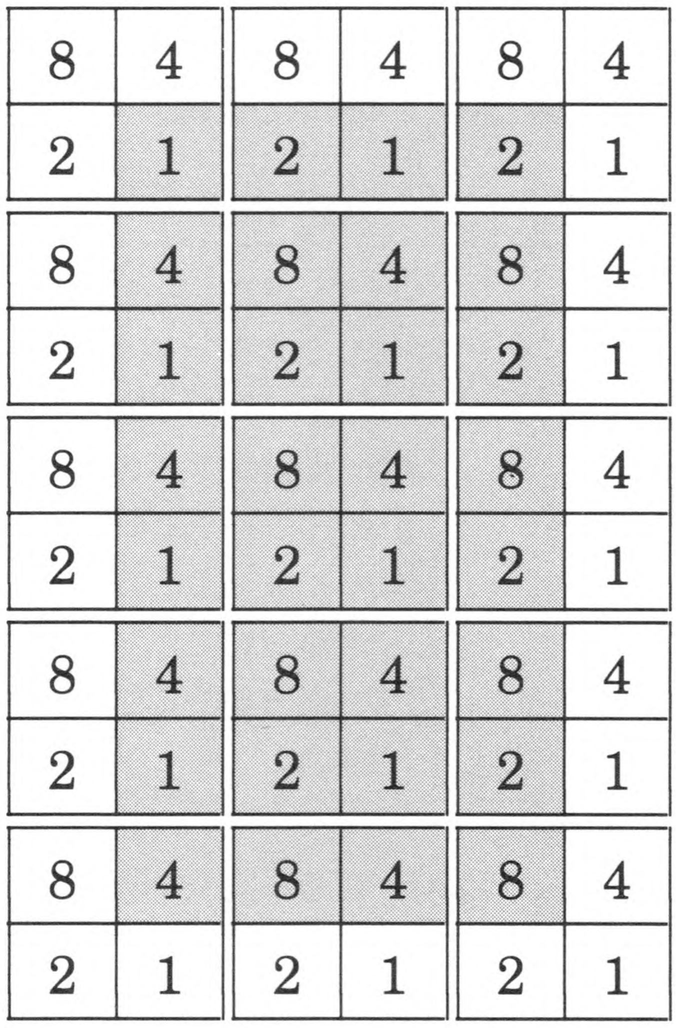
8	4
2	1

To calculate a character that has orange at Sections 8 and 1, add the section values to the first value in the orange group, 240, like this:

240 + 8 + 1 = 249

Character 249 is what you want.

The following diagram shows how you might block out a large letter O on the screen. The shaded portions of the characters are colored. The unshaded portions are black. In this case we want the colored portions to be green (the same color as the screen). You can do this using the color set 128 - 143.



Because Section 1 in the upper left character is to be colored, add 1 to the initial character value of 128. The first character value is 129. Moving right, Sections 2 and 1 are colored in the second character. Add 3 to 128 to get a second character value of 131. Calculate all 15 characters in this manner.

You could create a letter O in a BASIC09 procedure by *printing* each of the five rows of three characters. You could use DATA lines to store the ASCII codes for each character, then use loops to read and display the characters they represent.

Although low-level graphics is very rough, it can be useful, and it lets you mix graphics with text.

The following procedure not only creates the letter O, it adds the letter S and the number 9 to display the name of your operating system.

```
PROCEDURE os9prog
□DIM DAT:INTEGER
□PRINT CHR$(12)
□PRINT
□PRINT
□PRINT
□FOR Z=1 TO 5
□PRINT TAB(1Ø);
□FOR T=1 TO 12
□READ DAT
□PRINT CHR$(DAT);
□NEXT T
□PRINT
□NEXT Z
□END
□DATA 129,131,130,143,129,131,131,143,129,131,130,
143
□DATA 133,143,138,143,133,143,143,143,132,140,136,
143
□DATA 133,143,138,143,132,140,140,143,131,131,130,
143
□DATA 133,143,138,143,131,131,130,143,143,143,138,
143
□DATA 132,140,136,143,140,140,136,143,143,143,138,
143
```

Special Characters in High-Resolution

High-resolution graphics does not have graphic characters but it does have other international and special characters. These characters are represented by ASCII codes 128 through 159 as shown in the following table:

Table 9.4
High-Resolution Special Characters

Character	Hex Code	Decimal Code	Character	Hex Code	Decimal Code
Ç	80	128	ó	90	144
ü	81	129	æ	91	145
é	82	130	Æ	92	146
å	83	131	ô	93	147
ä	84	132	ö	94	148
à	85	133	ø	95	149
ā	86	134	û	96	150
ç	87	135	ù	97	151
ê	88	136	Ø	98	152
ë	89	137	Ŏ	99	153
è	8A	138	O	9A	154
ï	8B	139	§	9B	155
î	8C	140	£	9C	156
ß	8D	141	±	9D	157
Ä	8E	142	°	9E	158
Å	8F	143	*f*	9F	159

Medium-Resolution Graphics

For more sophisticated graphics operations, OS-9 has built-in graphics interface modules that provide a convenient way to access the graphics and joystick functions of the Color Computer 3. The required module for medium-resolution graphics is named GFX. It must be in your execution directory or resident in memory when called by BASIC09.

You can either install GFX in memory using the LOAD command, or wait until BASIC09 calls it for a graphics function. Once loaded, GFX resides in memory until you remove it using the OS-9 UNLINK command or the BASIC09 KILL command.

GFX has a number of functions that you pass to it as parameters with the RUN statement. For instance, the following statement clears the current graphics screen:

```
RUN GFX("CLEAR")
```

Other tasks need such parameters as position, color, and size. The following is a quick reference to all of the GFX functions. Each is explained in detail later:

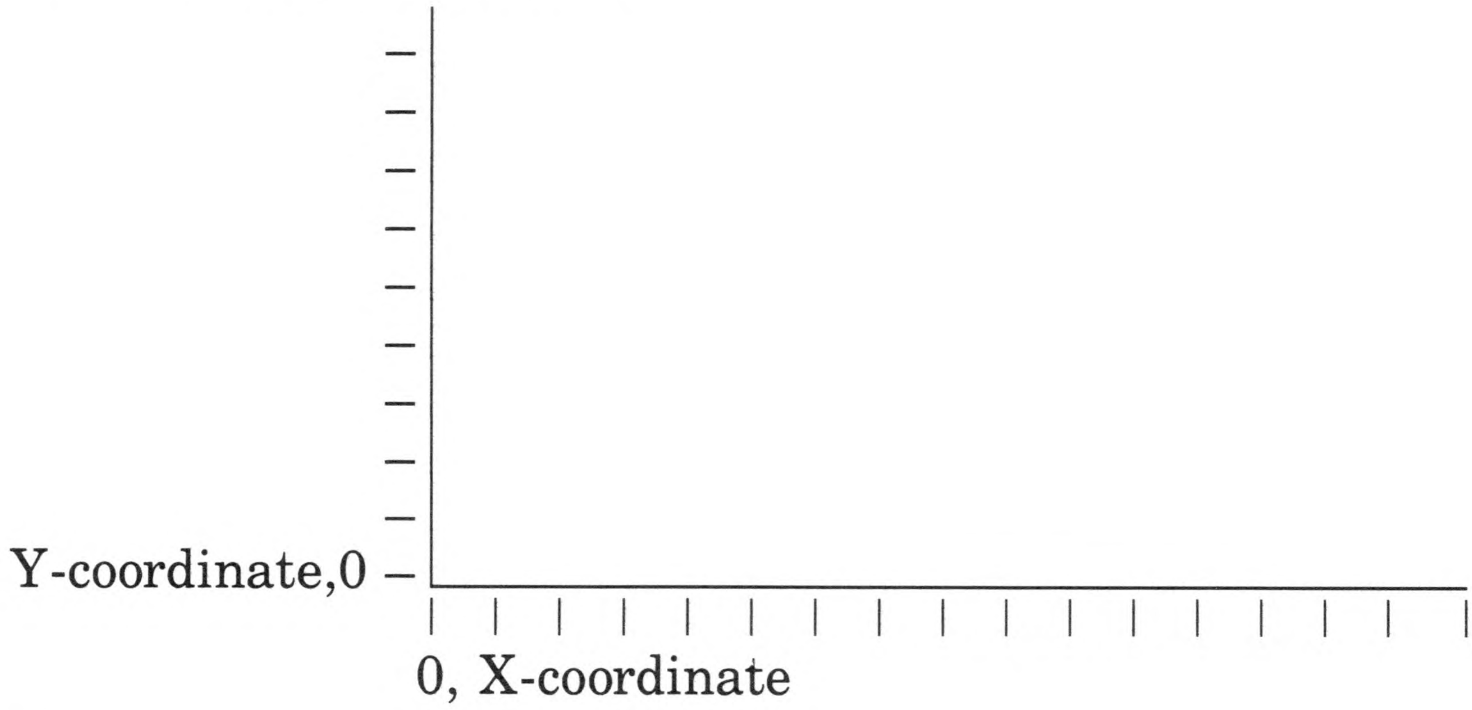
Function	Purpose	Parameters
ALPHA	Sets the screen to the alphanumeric mode.	None.
CIRCLE	Draws a circle.	Radius, optional X- and Y-coordinates, and color.
CLEAR	Clears the screen to a color.	Optional color for screen.
COLOR	Changes the foreground and background colors.	Foreground and background colors.
GCOLR	Reads a pixel's color.	Names of variables in which to store optional X- and Y-coordinates.
GLOC	Returns a video display address.	None.
JOYSTK	Returns the joystick button and X- and Y-coordinate status.	Names of variables in which to return the values.
LINE	Draws a line.	Ending X- and Y-coordinates, optional beginning coordinates, optional color.
MODE	Switches the screen between alphanumeric and graphics, sets the graphics screen color.	Format, Color.
MOVE	Positions the invisible graphics cursor.	X- and Y-coordinates.

Function	Purpose	Parameters
POINT	Moves graphics cursor and sets a point.	X- and Y-coordinates and optional pixel color.
QUIT	Returns screen to alphanumeric mode. Deallocates graphics memory.	None.

Formats and Colors

In medium-resolution graphics, you have a choice of two *formats*. Format 0 provides 256 horizontal points by 192 vertical points. In this format, you can have only two colors on the screen at a time.

Format 1 provides a 128 by 192 point screen and a maximum of four colors on the screen at a time. OS-9 medium-resolution graphics treats the screen as if it were a grid, with coordinate 0,0 at the lower left corner as shown in the following illustration. All points on the grid are positive.



BASIC09 defines colors with numbers or *color codes*. Many GFX functions allow or require color codes as parameters. BASIC09 also divides the color codes into *color sets*. Specifying a color code outside the current color set automatically initializes the new set.

Color Set	Color Code	Format 0 Background	Format 0 Foreground	Color Code	Format 1 Background	Format 1 Foreground
1	00	Black	Black	00	Green	Green
	01	Black	Green	01	Green	Yellow
	02			02	Green	Blue
	03			03	Green	Red
2	04	Black	Black	04	Buff	Buff
	05	Black	Buff	05	Buff	Cyan
	06			06	Buff	Magenta
	07			07	Buff	Orange
3				08	Black	Black
				09	Black	Dk Green
				10	Black	Md Green
				11	Black	Lt Green
4				12	Black	Black
				13	Black	Green
				14	Black	Red
				15	Black	Buff

Table 9.5

Use the preceding charts to chose colors for those functions that let you specify foreground or background colors. For instance, to initialize a Format 1 graphics screen with a green background and a red foreground, you type:

```
run gfx("mode",1,3)
```

The following reference section describes all the medium-resolution graphics functions, and provides examples and sample programs. To understand the organization of the commands reference, see "The Syntax Line" in Chapter 11.

The Draw Pointer

Medium-resolution graphics uses a *draw pointer*, or invisible graphics cursor, to determine what area of the screen is affected by graphics operations. When you establish a graphics screen, the draw pointer is located at coordinates 0,0. Some graphic functions automatically change the pointer location on the screen. For instance, the LINE function moves the draw pointer from the beginning coordinates to the end coordinates.

Because some functions begin at the draw pointer, you need to keep track of its location and make certain it is placed properly. Use the MOVE function to set the draw pointer to new locations.

ALPHA Select alphanumeric screen

Syntax: **RUN GFX("ALPHA")**

Function: Switches from the graphics screen to the alphanumeric (text) screen. The current graphics screen remains intact.

Parameters: None

Examples:

```
RUN GFX("ALPHA")
```

Sample Program:

This procedure lets you choose to draw a circle or rectangle of the size you select. Once you choose the shape and size, it uses the MODE function to select a graphics screen. When the shape is complete, you press ENTER to return to a text screen. The procedure uses the ALPHA function to return to the original menu.

```
PROCEDURE alpha
□DIM XCOR,YCOR,SIDE1,SIDE2,RADIUS,T,X,Y,Z:INTEGER
□DIM RESPONSE:STRING[1]
10 REPEAT
□SHELL "DISPLAY 0C"
□PRINT "Do you want to draw"
□PRINT "1) A rectangle"
□PRINT "2) A circle"
□PRINT "    -Press 1 or 2...";
□GET #0,RESPONSE
□PRINT
□IF RESPONSE="1" THEN
□INPUT "Length of Side 1...",SIDE1
□INPUT "Length of Side 2...",SIDE2
□RUN GFX("MODE",0,0)
□RUN GFX("CLEAR")
□XCOR=10
□YCOR=10
□RUN GFX("LINE",XCOR,YCOR,XCOR+SIDE1,YCOR,1)
```

```
□RUN GFX("LINE",XCOR+SIDE1,YCOR,XCOR+SIDE1,YCOR+
SIDE2,1)
□RUN GFX("LINE",XCOR+SIDE1,YCOR+SIDE2,XCOR,YCOR+
SIDE2,1)
□RUN GFX("LINE",XCOR,YCOR+SIDE2,XCOR,YCOR,1)
□INPUT RESPONSE
□ELSE
□IF RESPONSE="2" THEN
□INPUT "What radius?...",RADIUS
□RUN GFX("MODE",0,1)
□RUN GFX("CLEAR")
□RUN GFX("CIRCLE",128,90,RADIUS)
□INPUT RESPONSE
□ENDIF
□ENDIF
□UNTIL RESPONSE<>"1" AND RESPONSE<>"2"
□RUN GFX("ALPHA")
□GOTO 10
□END
```

CIRCLE **Draw a circle**

Syntax: **RUN GFX("CIRCLE"[,*xcor,ycor*],*radius* [,*color*])**

Function: Draws a circle of a given radius. If you do not specify a color, BASIC09 uses the current foreground color. If you do not specify X- and Y-coordinates, CIRCLE uses the current graphics cursor position as the circle's center.

Parameters:

radius	The radius of the circle you want to draw.
color	The code of the color you want the circle to be. See the chart earlier in this section for color information.
xcor,ycor	The X- and Y-coordinates for the center of the circle. Specifying coordinates outside the X-coordinate range of 0-255 or outside the Y-coordinate range of 0-191 causes an error.

Examples:

```
RUN GFX("CIRCLE",100)

RUN GFX("CIRCLE",100,3)

RUN GFX("CIRCLE",125,100,100)

RUN GFX("CIRCLE",125,100,100,2)
```

Sample Program:

This procedure uses CIRCLE to draw and erase a circle. The location of the circle changes before each draw/erase operation, causing the circle to move. When it hits the edge of the screen, it reverses its direction at a random angle and *bounces*.

```
PROCEDURE circles
□DIM RADIUS,XCOR,YCOR:INTEGER
□DIM XTEMP,YTEMP:INTEGER
□DIM PATH1,PATH2:INTEGER
□DIM FLAG:INTEGER
```

```
□FLAG=1
□XCOR=5
□YCOR=5
□PATH1=RND(15)+2
□PATH2=RND(10)+2
□XTEMP=249
□YTEMP=185
□RUN GFX("MODE",0,1)
□RUN GFX("CLEAR")
□FOR T=1 TO 200
□WHILE XCOR<250 AND XCOR>4 AND YCOR<186 AND YCOR>4
DO
□RUN GFX("CIRCLE",XTEMP,YTEMP,3,0)
□RUN GFX("CIRCLE",XCOR,YCOR,3,1)
□XTEMP=XCOR
□YTEMP=YCOR
□XCOR=XCOR+PATH1
□YCOR=YCOR+PATH2
□ENDWHILE
□PATH1=RND(15)+2
□PATH2=RND(10)+2
□IF XCOR>249 THEN
□XCOR=249
□ENDIF
□IF XCOR<5 THEN
□XCOR=5
□ENDIF
□IF YCOR>185 THEN
□YCOR=185
□ENDIF
□IF YCOR<5 THEN
□YCOR=5
□ENDIF
□FLAG=FLAG*-1
□IF FLAG<0 THEN
□PATH1=PATH1*-1
□PATH2=PATH2*-1
□ENDIF
□NEXT T
□END
```

CLEAR Clear the screen

Syntax: **RUN GFX("CLEAR"[,*color*])**

Function: Clears the current graphics screen. If you do not specify a color, CLEAR sets the entire screen to the current background color. CLEAR also sets the graphics cursor at coordinates 0,0, the lower left corner of the screen.

Parameters:

color A code indicating the color to set the screen.

Examples:

```
RUN GFX("CLEAR")

RUN GFX("CLEAR",14)
```

COLOR Change the foreground color

Syntax: **RUN GFX("COLOR",*color*)**

Function: Changes the foreground color (and possibly the color set). COLOR does not change the graphics format or the cursor position.

Parameters:

color A code indicating the color you want for the foreground. See the chart earlier in this chapter for color information.

Examples:

```
RUN GFX("COLOR",10)
```

Sample Program:

This procedure connects a series of differently colored circles to produce a necklace effect.

```
PROCEDURE necklace
□DIM COLOR,T,U,J,R,FLAG,XCOR,YCOR:INTEGER
□RUN GFX("MODE",1,0)
□RUN GFX("CLEAR")
□COLOR=1
□XCOR=1
□YCOR=1
□R=2
□FOR T=1 TO 6
□FOR J=1 TO 40
□XCOR=XCOR+1
□YCOR=YCOR+.8
□IF FLAG<0 THEN
□R=R-1
□ELSE
□R=R+1
□ENDIF
□COLOR=COLOR+1
□IF COLOR>3 THEN COLOR=1
```

```
□ENDIF
□RUN GFX("CIRCLE",XCOR,YCOR,R,COLOR)
□NEXT J
□FLAG=FLAG*-1
□NEXT T
□FOR U=1 TO 10000
□NEXT U
□END
```

GLOC **Find the graphics screen location**

Syntax: **RUN GFX("GLOC",*storage*)**

Function: Determines the location of the graphics screen in memory and returns the address in the specified variable. When you know the graphic screen address, you can use PEEK and POKE to perform special functions not available in the GFX module, such as filling a portion of the screen with a color or saving a graphics screen to disk.

OS-9 Level Two maps display screens into a program's address space before PEEK and POKE can operate on a display screen. This means that you must have at least eight kilobytes of free memory in the user's address space. Program and data memory requirements must not exceed 56 kilobytes.

Parameters:

storage	An integer or byte type variable in which GLOC stores the memory address of the graphics screen.

Examples:

```
RUN GFX("GLOC",location)
```

Sample Program:

This procedure uses the GLOC function to locate the current graphics screen, then uses POKE to *paint* a series of boxes on the screen.

```
PROCEDURE boxin
 DIM LOCATION,PLACE,COLOR,BEGIN,QUIT,X,TERMINATE,
LINE,T,J:INTEGER
 RUN GFX("MODE",1,0)
 RUN GFX("CLEAR")
 RUN GFX("GLOC",LOCATION)
 LOCATION=LOCATION+100 \ PLACE=LOCATION
 BEGIN=1
 QUIT=80
```

```
□COLOR=255
□TERMINATE=10
□LINE=32
□FOR X=1 TO 4
□FOR T=1 TO QUIT
□FOR J=BEGIN TO TERMINATE
□POKE PLACE+J,COLOR
□NEXT J
□PLACE=PLACE+LINE
□NEXT T
□LOCATION=LOCATION+160
□BEGIN=BEGIN+1
□PLACE=LOCATION
□QUIT=QUIT-10
□TERMINATE=TERMINATE-1
□COLOR=COLOR-85
□NEXT X
□INPUT Z$
□END
```

JOYSTK Get joystick status

Syntax: **RUN GFX("JOYSTK",*stick,fire,xcor,ycor*)**

Function: Determines the status of the specified joystick fire button and the X,Y position of the specified joystick handle. Use this function only with a standard joystick or mouse, not with the high-resolution mouse adapter.

Parameters:

stick	The joystick (0 or 1) for which you want to determine the status. 0 indicates the right joystick, 1 indicates the left joystick.
fire	A variable in which JOYSTK returns the status of the specified fire button. *Fire* can be byte, integer, or Boolean type. A value other than 0 or TRUE indicates the button is pressed.
xcor,ycor	Byte or integer type variables in which JOYSTK stores the X- and Y-coordinates of the joystick handle position. The coordinate range is 0-63.

Examples:

```
RUN GFX("JOYSTK",0,shoot,x,y)
```

Sample Program:

This procedure uses the JOYSTK function to draw on the screen with the right joystick.

```
PROCEDURE joydraw
□DIM STICK,FIRE,XCOR,YCOR,XTEMP,YTEMP:INTEGER
□RUN GFX("MODE",0,1)
□RUN GFX("CLEAR")
□JOY=0 \XCOR=0 \YCOR=0
□REPEAT
□XTEMP=XCOR
□YTEMP=YCOR
□RUN GFX("JOYSTK",0,FIRE,XCOR,YCOR)
□XCOR=XCOR*4
□YCOR=YCOR*4
□RUN GFX("LINE",XTEMP,YTEMP,XCOR,YCOR)
□UNTIL FIRE<>0
□END
```

LINE Draw a line

Syntax: **RUN GFX("LINE"[,*xcor1,ycor1*],*xcor2,ycor2* [,*color*])**

Function: Draws a line in the current or specified foreground color in one of the following ways:

- From the current draw position to the specified X,Y-coordinates.
- From the specified beginning X- and Y-coordinates to the specified ending X,Y-coordinates.

Parameters:

xcor1,ycor1	Are LINE's beginning X- and Y-coordinates.
xcor2,ycor2	Are LINE's ending X- and Y-coordinates.
color	A code indicating the color you want the line to be. See the chart earlier in this section for color information.

Examples:

```
RUN GFX("LINE",192,128)

RUN GFX("LINE",0,0,192,128)

RUN GFX("LINE",0,0,192,128,2)
```

Sample Program:

This procedure draws a sine wave of vertical lines across the screen.

```
PROCEDURE waves
□DIM A,B,C,D,X,Y,Z:INTEGER
□CALC=0 \ A=100
□RUN GFX("mode",0,1)
□RUN GFX("CLEAR")
□RUN GFX("COLOR",2)
□FOR X=0 TO 255 STEP 1
□CALC=CALC+.05
□Y=A-SIN(CALC)*15
□Z=Y+25
□RUN GFX("LINE",X,Y,X,Z)
□NEXT X
□END
```

MODE Switch to graphics screen

Syntax: **RUN GFX("MODE",*format,color*)**

Function: Switches the screen from alphanumeric (text) to graphics, selecting the screen format and color code. You must run MODE before you can use any other graphics function. When you do, BASIC09 allocates a six-kilobyte block of memory for graphics. If your system does not have this amount of memory available, OS-9 returns an error message.

Parameters:

format	Either 0 (a two-color 256 by 192 pixel screen) or 1 (a four-color, 128 by 192 pixel screen).
color	A code indicating the color to set the screen. See the chart earlier in this chapter for information on color sets.

Examples:

```
RUN GFX("MODE",1,2)
```

MOVE **Move graphics cursor**

Syntax: **RUN GFX("MOVE",*xcor,ycor*)**

Function: Moves the invisible graphics cursor to the specified location on the screen. MOVE does not change the display in any way.

Parameters:

xcor,ycor	The coordinates for the cursor.

Examples:

```
RUN GFX("MOVE",192,128)
```

Sample Program:

This procedure draws and *pops* bubbles on the screen using the CIRCLE function. It uses MOVE to select the position for the circles.

```
PROCEDURE bubbles
  DIM XCOR,YCOR,T,R,ARRAY(3,100):INTEGER
  RUN GFX("MODE",1,0)
  RUN GFX("CLEAR")
  FOR T=1 TO 20
  ARRAY(1,T)=RND(255)
  ARRAY(2,T)=RND(192)
  ARRAY(3,T)=RND(50)
  RUN GFX("MOVE",ARRAY(1,T),ARRAY(2,T))
  RUN GFX("CIRCLE",ARRAY(3,T),3)
  NEXT T
  FOR T=1 TO 20
  RUN GFX("MOVE",ARRAY(1,T),ARRAY(2,T))
  RUN GFX("CIRCLE",ARRAY(3,T),0)
  SHELL "DISPLAY 07"
  NEXT T
  END
```

POINT Set point to specified color

Syntax: **RUN GFX("POINT",*xcor,ycor*[,*color*])**

Function: Displays a dot at the specified coordinates. If you specify a color, POINT sets the pixel at the new coordinates to that color. Otherwise, POINT sets the pixel at the new coordinates to the foreground color.

Parameters:

xcor,ycor The X- and Y-coordinates for a pixel.

color The code of the color you want the pixel to be. See the chart earlier in this section for color information.

Examples:

```
RUN GFX("POINT",192,128)

RUN GFX("POINT",192,128,2)
```

Sample Program:

This procedure uses POINT to draw filled boxes on the screen.

```
PROCEDURE boxup
□DIM XCOR,YCOR,BEGIN,COLOR,QUIT,TERMINATE,LINE:
INTEGER
□DIM T,X,Y:INTEGER
□XCOR=50 \YCOR=30 \COLOR=1
□BEGIN=1 \START=1 \QUIT=20 \TERMINATE=50
□RUN GFX("MODE",1,0)
□RUN GFX("CLEAR")
□FOR T=1 TO 4
□FOR X=BEGIN TO QUIT
□FOR Y=START TO TERMINATE
□RUN GFX("POINT",XCOR+Y,YCOR,COLOR)
□NEXT Y
□YCOR=YCOR+1
□NEXT X
□START=START+10
```

```
□TERMINATE=TERMINATE-10
□COLOR=COLOR+1
□NEXT T
□INPUT Z$
□END
```

QUIT Deallocate graphics screen

Syntax: **RUN GFX("QUIT")**

Function: Switches the screen to the alphanumeric (text) mode and deallocates graphics memory.

Parameters: None

Examples:

```
RUN GFX("QUIT")
```

High-Resolution Graphics

BASIC09's high-resolution graphics greatly expand the capabilities of the Color Computer 3. You can have greater screen resolution (up to 640 by 192 pixels), as many as 64 colors, and the ability to mix graphics and text on one screen. In addition, you can use different text *fonts*, or styles.

The high-resolution module, GFX2, has many more functions than its medium resolution counterpart. GFX2 gives you the ability to:

- Select from 64 colors. OS-9 provides a palette with 16 default colors. You can change any of these default colors to any of the 64 colors available on the Color Computer 3.
- Set border colors.
- Set color patterns.
- Create different types of graphics screen cursors.
- Use logic functions.
- Turn an automatic scaling function off or on.
- Draw outline or filled boxes.
- Draw ellipses and arcs.
- Fill specified areas with specified colors.
- GET and PUT sections of the graphics screen.
- Select character fonts, which include boldfaced, transparent, and proportionally spaced characters.
- Move the cursor. Erase portions of a line or of the screen.
- Select reverse or normal video.
- Underline text.

Also, high-resolution graphics operate through the OS-9 Windowing System. This means that you can run several procedures in different *windows*. You can establish windows to display text, or to display graphics, or both. You can easily display any window.

Establishing a Hardware Window

For your convenience, OS-9 has a number of predefined or *hardware* window formats. Hardware windows are text windows, and you cannot use them for graphic applications. Because hardware windows are predefined, you can easily establish them with the INIZ command. For instance, to establish Window 7, type:

```
iniz w7 [ENTER]
```

However, you cannot see the window until you send a message to it. Type:

```
echo Hello Window 7 > /w7 [ENTER]
```

Now, to see the window and your message press [CLEAR]. To return to the original screen, press [CLEAR] again.

To OS-9, a window is a device and you can send data to it. To view the Errmsg file in the SYS directory of your system diskette, list it to Window 7 by typing:

```
list sys/errmsg > /w7 [ENTER]
```

Press [CLEAR] to move to Window 7 and see the listing. Press [SHIFT][CLEAR] to return to the previous screen.

You can also fork a shell (an execution environment) to a window. To cause a shell to operate in Window 7, type:

```
shell i=/w7& [ENTER]
```

The `i=` function of SHELL tells OS-9 that the window is *immortal*. It does not die after completing a task. To operate OS-9 from the window, press [CLEAR].

Besides Window 7, you have six other predefined windows. The following chart shows all the hardware windows and their parameters:

Window Number	Screen Size Chars/line	Starting Coordinates X-Coord, Y-Coord	Window Size Cols	Rows
1	40	0,0	27	11
2	40	28,0	12	11
3	40	0,12	40	12
4	80	0,0	60	11
5	80	60,0	19	11
6	80	0,13	80	12
7	80	0,0	80	24

Defining Windows

As well as hardware windows, OS-9 also lets you establish windows to your own specifications. You can set definable windows for either text or graphics, or both. You can locate them anywhere on a screen, and you can make them any size.

You initialize definable windows in the same manner you initialize hardware windows, using INIZ. If you want to have text on the window, you must merge SYS/Stdfonts (found on your system diskette) with the window. You can also establish a shell in a definable window, from which you can use OS-9 or BASIC09.

To establish definable windows you must supply OS-9 with information about the type of window you want (its graphic format), its size, and its location on the screen. The easiest way to do this is with the OS-9 WCREATE command.

WCREATE requires a window format code in the form `- s=format code` to tell OS-9 what type of a window you want. The following chart shows the possible window formats you can choose:

Table 9.6

Format Code	Screen Size Cols x Rows	Resolution Width/Height	No. of Colors	Memory Required	Screen Type
01	40 x 24	——	16†	1600	Text
02	80 x 24	——	16†	4000	Text
05	80 x 24	640 x 192	2	16000	Graphics
06	40 x 24	320 x 192	4	16000	Graphics
07	80 x 24	640 x 192	4	32000	Graphics
08	40 x 24	320 x 192	16	32000	Graphics
00*	Specifies the current screen.				
FF	Current display screen. Use when putting several windows on the same physical screen.				

† You have to reconfigure the palette to get 16 colors rather than the default of eight colors. The following section provides information on the palette.

Format Codes 01 and 02 select text screens, and Format Codes 5-8 select graphics screens. The Screen Size column shows the maximum number of text columns and rows available for each screen. The Resolution column shows the maximum *pixels* (graphic units) available for each of the graphic screens. The Memory column shows how much memory OS-9 must set aside for each screen format. Memory requirements depend on the resolution and number of colors selected for a window.

The Palette

BASIC09 has 64 colors you can select for screen displays. The colors are available through a *palette*. The Color Computer's palette can hold 16 colors at once.

The following chart shows the default colors for the palette in Screen Format 7:

Table 9.7

Register	Color	Register	Color
00	Black	**08**	Black
01	Red	**09**	Green
02	Green	**10**	Black
03	Yellow	**11**	Buff
04	Blue	**12**	Black
05	Magenta	**13**	Green
06	Cyan	**14**	Black
07	White	**15**	Orange

Instead of the default colors, you can select any of the 64 colors (0-63) for any of the palette registers. You do this using the PALETTE command described later in this chapter. The BORDER and COLOR commands also affect the colors available in the palette by changing the color in the background and foreground registers, Registers 02 and 03, respectively.

> **Note:** The information in the next section assumes you have a Color Computer 3 with 512 kilobytes of memory. If your computer has 128 kilobytes of memory, skip to the section "High-Level Graphics With 128K."

Establishing a Graphics Window

To create any window, you should first initialize it with the INIZ command. Type:

```
iniz w1 [ENTER]
```

So that you can later type in the new window, merge the Stdfonts file with it. Type:

```
merge sys/stdfonts>/w1 [ENTER]
```

Using the information in the preceding tables, use WCREATE to establish a graphics window. The following command line creates a graphics window in Window 1 that has 320 x 192 resolution and that fills the entire screen. The new window has 16 colors available and provides 40 column by 24 line text:

```
wcreate /w1 -s=8 00 00 40 24 03 02 02 [ENTER]
```

The screen border color

The screen background color

The screen foreground color

The screen length in rows

The screen width in columns

The Y-Coordinate for the beginning of the screen

The X-coordinate for the beginning of the screen

The screen type

The window name

The command name

Starting a Shell in a Window

At this point, the new window exists, and you can send data to it. However, if you want to operate from the window, you must install a shell in it. Type:

```
shell i=/w1& [ENTER]
```

Press [CLEAR] to move to the new window. To load BASIC09, type:

```
basic09 #10K [ENTER]
```

Select either more or less memory, according to your needs. Using BASIC09 in a graphics window, you can write procedures to create high-resolution graphics, and you can display the graphics on the same screen.

Using High-Level Graphics With 128K

If your computer is equipped with only 128 kilobytes of memory, you cannot use more than one window with BASIC09. Also, to use even one window, you must follow certain steps to provide enough memory for BASIC09 operations.

Refer to Table 9.6. You must select a window mode that does not use more than 16000 byte of memory—either window Format 5 or Format 6.

To provide enough memory to use BASIC09, you must fork a shell to the window you create, then kill the shell in TERM. Doing this means that you can no longer operate from your TERM screen. However, you can run OS-9 and BASIC09 from the window.

The following steps show you how to create a Format 6 graphics screen in Window 1, write a BASIC09 high-resolution graphics procedure, and execute it using minimum memory.

1. Boot OS-9. Then, create a graphics window by typing:

```
iniz w1 [ENTER]
wcreate /w1 -s=06 00 00 40 24 06 01 01 [ENTER]
merge sys/stdfonts>/w1 [ENTER]
shell i=/w1& [ENTER]
ex [ENTER]
```

2. The system stops, and you can no longer type or issue commands. Press [CLEAR] to move to the new window. Then, load BASIC09 by typing:

```
basic09 [ENTER]
```

3. Enter the edit mode, and type the following procedure:

```
PROCEDURE squeeze
DIM XCOR,YCOR,X,Y:INTEGER; RESPONSE:STRING[1]
RUN GFX2("CUROFF")
XCOR=320 \ YCOR=95 \ X=300 \ FLAG=1
PRINT CHR$(12)
LOOP
FOR Y=1 TO 100 STEP 2
X=X-3
GOSUB 10
IF FLAG<1 THEN
RUN GFX2("COLOR",0)
```

```
□ELSE
□RUN GFX2("COLOR",3)
□ENDIF
□RUN GFX2("ELLIPSE",XCOR,YCOR,X,Y)
□FLAG=FLAG*-1
□NEXT Y
□RUN GFX2("COLOR",1)
□FOR Y=99 TO 1 STEP -2
□GOSUB 10
□X=X+3
□RUN GFX2("ELLIPSE",XCOR,YCOR,X,Y)
□NEXT Y
□RUN GFX2("COLOR",0)
□ENDLOOP
10□RUN INKEY(RESPONSE)
□IF RESPONSE="" THEN
□RETURN
□ENDIF
10□PRINT CHR$(12)
□RUN GFX2("COLOR",0)
□RUN GFX2("CURON")
□END
```

4. When you have entered the procedure exactly as shown, exit the edit mode, and from the BASIC09 command mode, save Squeeze by typing:

   ```
   save squeeze [ENTER]
   ```

5. Compile Squeeze by typing:

   ```
   pack squeeze [ENTER]
   ```

   Squeeze is now an executable module saved in your current execution directory. The following steps assume your execution directory is /D0/CMDS.

6. Exit BASIC09 by typing:

   ```
   bye [ENTER]
   ```

7. Merge Squeeze, RUNB, INKEY, and GFX2 into one module. To do this, type:

   ```
   merge /d0/cmds/squeeze /d0/cmds/runb /d0/cmds/
   inkey gfx2 > /d0/cmds/yawn [ENTER]
   ```

8. MERGE does not set the new file Yawn as an executable file. Before you execute it, you must make the file executable by typing:

   ```
   attr /d0/yawn e pe [ENTER]
   ```

9. To execute Yawn, type:

   ```
   yawn [ENTER]
   ```

10. To terminate the procedure, press the space bar.

The merging procedure in Step 7 saves a considerable amount of memory. Every module you load uses one or more 8-kilobyte blocks of storage space. For instance, INKEY is only 94 bytes in length. However, if you load it as a separate module, it requires 8192 bytes. RUNB is 12185 bytes in length. This means that it requires two 8-kilobyte blocks, or 16384 bytes of memory. GFX2 is 2190 bytes in length, and Squeeze is 605 bytes in length. Loaded individually, they also require two memory blocks.

If you load all four modules independently, they use 40960 bytes. However, by combining them into one file, they load into two memory blocks, or 16384 bytes.

Using the information in this section, you can write and execute numerous BASIC09 procedures with only 128 kilobytes of memory. However, if your computer has 512 kilobytes of memory, you can bypass many of these steps. Also, the additional memory enables you to have several windows open at one time. For instance, you can create one window in which to write BASIC09 procedures, another window in which to execute your procedures, and a third window from which you can use OS-9 commands.

> **Note:** The remainder of this chapter assumes you have 512 kilobytes of memory. If you don't, you can still run many of the sample procedures by implementing the steps in this section.

Creating Windows from BASIC09

Using GFX2 routines, BASIC09 provides the means to create and manage windows. The steps for creating windows from BASIC09 are as follows:

1. DIM a variable to hold the path number to the window you want to create.

2. OPEN a path to the window.
3. SELECT the new window as the display window.
4. Send commands, data, or text to the window through the open path.
5. CLOSE the open path.
6. Use SELECT to return to your original window.

If you do not want to return immediately to the screen or window of origin, you can skip Steps 5 and 6.

The following sample procedure shows how to open Window 2 as a 320 x 192 graphics window, draw a circle, then return to the original screen when you press a key.

```
PROCEDURE make_win
DIM PATH:INTEGER
DIM RESPONSE:STRING[1]
OPEN #PATH,"/W2":WRITE
RUN GFX2 (PATH,"DWSET",08,00,00,40,24,03,02,02)
RUN GFX2 (PATH,"SELECT")
RUN GFX2 (PATH,"CIRCLE",200,90,80)
GET #1,RESPONSE
CLOSE #PATH
RUN GFX2 ("SELECT")
END
```

This procedure establishes a Format 8 window, beginning at Coordinates 0,0 and covering the total screen. The foreground color is green, the background color is black, and the border color is black.

Because this procedure does not INIZ the window it opens, the window automatically disappears when the procedure closes its path. To create a window that stays in the system, even after you close the path to it, use INIZ before the OPEN statement, like this:

```
SHELL "INIZ /W2"
```

After you create and define the window, view it by pressing CLEAR. To get back to the screen you are working on, press SHIFT CLEAR. If you intend to use a window more than once in a procedure, you do not need to close its path until the procedure no longer needs it.

Creating Overlay Windows

When you establish a window, you are initializing an OS-9 device. However, an overlay window is only a new screen for an existing window. An overlay screen can be the same size as its window, or it can be smaller. OS-9 automatically transfers to the overlay window any current procedures operating in the device window.

The process for creating overlay windows lets you select whether you want to save the contents of the screen covered by the new window. If you choose to save the contents, the previous screen is redisplayed when you end the overlay.

The following procedure provides an example of using overlay windows. It creates six overlays, each smaller than the preceding window. The procedure then waits for you to press a key. When you do, it removes the overlay windows.

```
PROCEDURE overwindows
□DIM X,Y,X1,Y1,T,J,B,L,PLACE:INTEGER
□DIM RESPONSE:STRING[1]
□X=0 \Y=0
□X1=80 \Y1=24
□PLACE=33
□FOR T=1 TO 6
□IF T=2 OR T=6 THEN
□B=3
□ELSE B=2
□ENDIF
□RUN GFX2("OWSET",1,X,Y,X1,Y1,B,T)
□X=X+6 \Y=Y+2
□X1=X1-12 \Y1=Y1-4
□FOR J=1 TO 5
□PRINT TAB(PLACE); "Overlay Screen "; T
□NEXT J
□PLACE=PLACE-6
□NEXT T
□PRINT "Overlay Screen 6"
□PRINT "Press A Key...";
□GET #1,RESPONSE
□FOR T=1 TO 6
□RUN GFX2("OWEND")
□NEXT T
□END
```

The Graphics Cursor and the Draw Pointer

High-resolution graphics provide a text cursor, a graphics cursor, and a *draw pointer*. The text cursor and the graphics cursor can be either visible or invisible. The draw pointer is always invisible.

Text functions always begin at the current location of the text cursor. Whenever you *print* on the screen, the cursor automatically moves to the end of the text or to the beginning of the next line, depending on whether or not you use a semicolon after the print statement. You can reset the text cursor to any place on the screen with the CURXY function of GFX2.

Many BASIC09 graphics functions also begin operating at a location pointed to by the draw pointer. When you begin graphics, the draw pointer is located at coordinates 0,0. BASIC09 then updates the pointer as you execute certain graphics functions. For instance, the LINE function of GFX2 draws from the draw pointer position to the specified end coordinates. The draw pointer is left pointing to the end coordinates.

Because some functions begin at the draw pointer, you need to keep track of its location and make certain it is placed properly. Use the SETDPTR function to move the draw pointer to new locations.

The graphics cursor is for use with joystick or mouse operations. It provides a *pointer* for graphics applications. The system diskette provides patterns that can be loaded into the graphics cursor *buffer*. You can select from a variety of pointer images.

High-Resolution Text

When you create a graphics window, you can display either text characters, graphics characters, or both.

To display graphics, move the draw pointer to the location where you want the graphics to begin. Then, execute the graphics routines.

To display text, move the text cursor to the location where you want the text to begin. Then, use normal BASIC commands to *print* text.

Instructions for the draw pointer relate to a 640 x 192 grid, numbered 0-639 and 0-191. Instructions for the text cursor relate to the number of characters per line and the number of lines on the current screen format.

Using Fonts

OS-9 has built-in fonts (character sets). You can also create your own fonts and instruct BASIC09 to use them. If you create your own fonts, you can design any symbols or graphics characters you want to use.

To use fonts, you must be in a graphics window. See "Establishing a Graphics Screen" earlier in this chapter. Use the FONT function to tell OS-9 what font you want. BASIC09 has three fonts installed in Group 200, Buffers 1, 2, and 3. The following procedure uses characters in Buffer 3 to draw a border, then prints a message using the characters in Buffer 2. It then returns to Buffer 3 and asks you to press a key to end the procedure.

```
PROCEDURE borders
 DIM T,B,V,J,K:INTEGER
 DIM RESPONSE:STRING[1]
 B=199
 PRINT CHR$(12)
 RUN GFX2("FONT",200,3)
 RUN GFX2("COLOR",1,2)
 FOR T=0 TO 79
 PRINT CHR$(B);
 NEXT T
 FOR T=1 TO 21
 RUN GFX2("CURXY",0,T)
 PRINT CHR$(B); CHR$(B);
 RUN GFX2("CURXY",78,T)
 PRINT CHR$(B); CHR$(B);
 NEXT T
 RUN GFX2("CURXY",0,21)
 FOR T=0 TO 79
 PRINT CHR$(B);
 NEXT T
 RUN GFX2("FONT",200,2)
 RUN GFX2("COLOR",0,2)
 RUN GFX2("CURXY",45,9)
 PRINT "A Demonstration"
```

```
□RUN GFX2("CURXY",50,10)
□PRINT "Of A"
□RUN GFX2("CURXY",43,11)
□PRINT "Buffer Three Border"
□RUN GFX2("CURXY",51,12)
□PRINT "And"
□RUN GFX2("CURXY",45,13)
□PRINT "Buffer Two Text"
□RUN GFX2("FONT",200,1)
□RUN GFX2("COLOR",3,2)
□RUN GFX2("CURXY",33,15)
□PRINT "Press A Key...";
□GET #1,RESPONSE
□PRINT CHR$(12)
□END
```

High-Resolution Quick Reference

High-resolution functions are all part of the GFX2 module. You call them in a BASIC09 procedure with the following syntax:

RUN GFX2([*PATH*],"*FUNCTION*"[,*PARAMETER*[,...]])

Path is an optional variable name that tells OS-9 the window in which you want the function performed. *Function* is the high-resolution task you want to perform. *Parameter* is an essential or optional value that affects the performance of the function. Different functions require or permit different numbers of parameters.

The following reference gives a brief description of the high-resolution graphics functions. This list is organized by function. Following the quick reference is a detailed reference organized alphabetically.

Window Commands

Command	Function
DWSet	Establishes a window and sets its location on the screen, its size, its background color, its foreground color, and its border color.
OWSet	Establishes an overlay window on a device window that already exists. The function also sets the overlay window size, background color, foreground color, and border color. When using this function, you can choose whether or not to save the contents of the original screen.
OWEnd	Deallocates the specified overlay window.
Select	Selects the window to display.
DWEnd	Deallocates an established window.
CWArea	Changes the size of a window. You can only reduce the working area of a window, not increase it.
DWProtectSw	Lets you unprotect a window and set other device windows over it. This might destroy the contents of either or both windows.

Drawing Commands:

Command	Function
Point	Sets the pixel under the draw pointer to the specified color or to the default color.
Line	Draws a line.
Box	Draws a rectangle outline.
Bar	Draws a filled rectangle.
Circle	Draws a circle.
Ellipse	Draws an ellipse.
Arc	Draws an arc.
Fill	Fills the area of the window the same color as the pixel under the draw pointer.
Clear	Clears the window.

Configuring Commands:

Command	Function
Color	Sets any of the foreground, background, or border colors.
DefCol	Sets palette registers to the default colors.
Border	Sets the border palette register.
Palette	Changes colors in the palette registers.
Pattern	Establishes a buffer from which BASIC09 gets a pattern for graphics functions.
Logic	Turns on AND, OR, or XOR logic functions for draw functions.
GCSet	Establishes a buffer from which BASIC09 gets the graphics cursor.
ScaleSw	Turns scaling on or off.
SetDPtr	Positions the draw pointer.
PutGC	Positions the graphics cursor.
Draw	Draws an image from directions provided in a draw string.

Get/Put Commands:

Command	Function
Get	Saves a specified portion of a window to a buffer.
Put	Places the image stored in a buffer onto a window.
DefBuff	Defines a buffer for storage.
GPLoad	Preloads a buffer from a disk file.
KillBuff	Deallocates a buffer.

Text/Cursor Handling Routines:

Command	Function
CurHome	Positions the cursor at coordinates 0,0.
CurXY	Positions the cursor at specified coordinates.
ErLine	Erases the line under the cursor.
ErEOLine	Erases from the cursor to the end of the line.
CurOff	Turns the graphics cursor off.
CurOn	Turns the graphics cursor on.
CurRgt	Moves the graphics cursor right one space.
Bell	Sounds the terminal bell.
CurLft	Moves the graphics cursor left one space.
CurUp	Moves the graphics cursor up one line.
CurDwn	Moves the graphics cursor down one line.

Font Handling Commands:

Command	Function
Font	Specifies the buffer from which BASIC09 selects its font characters.
TCharSw	Selects or deselects transparent characters.
BoldSw	Selects or deselects bold characters.
PropSw	Selects or deselects proportional characters.
ErEoWndw	Erases from the graphics cursor to the end of the window.
Clear	Erases window and homes the cursor.
CrRtn	Performs a carriage return by moving the cursor down one line and to the extreme left of the window.
ReVOn	Turns reverse video on.
ReVOff	Turns reverse video off.
UndlnOn	Turns the underline function on.
UndlnOff	Turns the underline function off.
BlnkOn	Turns blinking characters on (only for hardware text screens).
BlnkOff	Turns blinking characters off (only for hardware text screens).
InsLin	Inserts a blank line at the graphics cursor position.
DelLin	Deletes the line at the graphics cursor position.

ARC Draw an arc

Syntax: **RUN GFX2([*path*,]"ARC"[,*mx*,*my*], *xrad*,*yrad*,*xcor1*,*ycor1*,*xcor2*, *ycor2*)**

Function: Draws an arc at the current or specified draw position with the specified X and Y radius. If you specify the same radius for both X and Y, the function draws a circular arc, otherwise the arc is elliptical. The X coordinates are in the range 0-639. The Y coordinates are in the range 0-191.

ARC begins drawing from the point on the screen closest to the first set of coordinates (xcor1, ycor1). It stops at the portion of the screen closest to the second set of coordinates (xcor2, ycor2). You can determine on which side of the line ARC draws by selecting which set of coordinates is the beginning and which set is the end.

Parameters:

path	The route to the window in which you want to draw an arc.
mx,my	The X- and Y-coordinates for the center of the arc. If you do not specify *mx* and *my*, BASIC09 uses the current draw pointer position.
xrad	The radius of the arc's width.
yrad	The radius of the arc's height.
xcor1,ycor1 *xcor2,ycor2*	The beginning and ending coordinates for an imaginary line from which the function draws an arc. The line is relative to the center of the arc (the center point is at 0,0 for these coordinates) and extends through the two coordinates from one edge of the screen to the other.

Examples:

```
RUN GFX2("ARC",50,100,50,100,50,150)
```

Sample Program:

This procedure draws a series of diagonally-cut arcs on a graphics window screen.

```
PROCEDURE arcing
□DIM MX,MY,XRAD,YRAD,XCOR,YCOR,XCOR2,YCOR2:INTEGER
□DIM T,X,Y,Z:INTEGER
□PRINT CHR$(12)
□FOR T=1 TO 90 STEP 2
□RUN GFX2("ARC",318,95,150,T,0,1,0,1)
□RUN GFX2("ARC",324,95,150,T,1,0,1,1)
□NEXT T
```

BAR Fill a rectangle

Syntax: **RUN GFX2([*path*,]"BAR"[,*xcor1,ycor1*],*xcor2, ycor2*)**

Function: Fills a rectangular area defined by two sets of coordinates. BAR defines its area with an imaginary diagonal line from the first set of coordinates to the second set of coordinates. The X coordinates are in the range 0-639. The Y coordinates are in the range 0-191.

Parameters:

path	The route to the window in which you want to draw a bar.
xcor1,ycor1	The beginning coordinates of the line defining the area to fill. If you omit these coordinates, BAR uses the draw pointer position. See the previous section "The Graphics Cursor and The Draw Pointer." Also see SETDPTR.
xcor2,ycor2	The ending coordinates of the line defining the area to fill.

Examples:

```
RUN GFX2("BAR",200,100)

RUN GFX2("BAR",0,0,100,50)
```

Sample Program:

This procedure draws a bar chart on a window screen.

```
PROCEDURE OSgraf
 DIM COLOR,T,X,XCOR1,YCOR1,XCOR2,YCOR2:
INTEGER; RESPONSE:STRING[1]
 PRINT CHR$(12)
 RUN GFX2("DEFCOL")
 COLOR=13 \ XCOR1=10 \ YCOR1=180
 XCOR2=XCOR1+40
 RUN GFX2("CUROFF")
```

```
□FOR T=1 TO 10
□READ YCOR2
□RUN GFX2("COLOR",COLOR)
□RUN GFX2("BAR",XCOR1,YCOR1,XCOR2,YCOR2)
□RUN GFX2("COLOR",7)
□RUN GFX2("BOX",XCOR1,YCOR1,XCOR2,YCOR2)
□COLOR=COLOR+1 \ XCOR1=XCOR1+50 \ XCOR2=XCOR1+40
□NEXT T
□PRINT \ PRINT "    OS-9 Sales Chart"
□RUN GFX2("BOX",0,0,510,180)
□GET #1,RESPONSE
□RUN GFX2("CURON")
□PRINT CHR$(12)
□END
□DATA 170,150,140,130,110,90,70,60,50,30
```

BELL Ring the terminal bell

Syntax: **RUN GFX2("BELL")**

Function: Rings the terminal's *bell* (produces a beep through the speaker).

Parameters: None

Examples:

```
RUN GFX2("BELL")
```

BLNKON Character blink on
BLNKOFF Character blink off

Syntax: **RUN GFX2([*path*,]"BLNKON")**
RUN GFX2([*path*,]"BLNKOFF")

Function: Executing BLNKON causes all subsequent characters sent to a window on a *hardware* screen to blink. A hardware screen is one of the predefined device windows /W1 through /W7. Executing BLNKOFF cancels a previous blink command; characters already blinking continue to do so. Blink does not operate on graphics windows.

Parameters:

path The route to the window in which you want to blink characters.

Examples:

```
RUN GFX2("BLNKON")

RUN GFX2("BLNKOFF")
```

BOLDSW **Switch bold characters on or off**

Syntax: **RUN GFX2([*path*,]"BOLDSW","*switch*")**

Function: Causes characters to display in either regular or bold typeface. The default is regular typeface. BOLD only works on graphics screens.

Parameters:

path — The route to the window in which you want bold characters.

switch — Can be either "ON" or "OFF." If *switch* is "ON," subsequent characters are bold. If *switch* is "OFF," subsequent characters are not bold.

Examples:

```
RUN GFX2("BOLDSW","ON")
```

Sample Program:

This procedure demonstrates the BOLDSW function by displaying both bold and normal text on a window screen.

```
PROCEDURE bold
□DIM LINE:STRING
□DIM LETTER:STRING[1]
□DIM T,J,K,FLAG:INTEGER
□RUN GFX2("CLEAR")
□FLAG=1
□FOR T=1 TO 8
□READ LINE
□FOR J=1 TO LEN(LINE)
□LETTER=MID$(LINE,J,1)
□IF LETTER<>"!" AND LETTER<>"#" THEN
□PRINT LETTER;
□ENDIF
□IF LETTER="!" THEN
□FLAG=FLAG*-1
```

```
□IF FLAG>0 THEN
□RUN GFX2("BOLDSW","OFF")
□ELSE
□RUN GFX2("BOLDSW","ON")
□ENDIF
□ENDIF
□IF LETTER="#" THEN
□PRINT CHR$(34);
□ENDIF
□NEXT J
□PRINT
□NEXT T
□PRINT \ PRINT
□END
□DATA "This is a demonstration of"
□DATA "the !Bold! function of"
□DATA "BASIC09's GFX2 module."
□DATA "Use the command"
□DATA "!RUN GFX2(#BOLDSW#,#ON#)!"
□DATA "to turn boldface on."
□DATA "Use !RUN GFX2(#BOLDSW#,#OFF#)!"
□DATA "to turn boldface off"
```

BORDER Set the border color

Syntax: **RUN GFX2([*path*,]"BORDER",*color*)**

Function: Resets the palette register that affects a window's border color (Register 0) to the specified color code. For information on the palette and on screen colors, see "The Palette" and Table 9.7 earlier in this chapter.

Parameters:

path	The route to the window in which you want to change border color.
color	One of the current palette colors. *Color* can be either a constant or a variable.

Examples:

```
RUN GFX2("BORDER",1)
```

Sample Program:

This procedure lets you select different border colors by pressing [+] or [–] to select higher or lower color codes. Press [q] to end the procedure.

```
PROCEDURE border
□DIM COLOR:INTEGER
□DIM KEY:STRING[1]
□COLOR=8
□RUN GFX2("CLEAR")
□WHILE KEY<>"q" AND KEY<>"Q" DO
□GET #1,KEY
□IF KEY="-" OR KEY="=" THEN
□COLOR=COLOR-1
□ENDIF
□IF KEY="+" OR KEY=";" THEN
□COLOR=COLOR+1
□ENDIF
```

```
□IF COLOR>15 OR COLOR<0 THEN COLOR=8
□ENDIF
□RUN GFX2("BORDER",COLOR)
□RUN GFX2("CURXY",0,0)
□ENDWHILE
□END
```

BOX Draw a rectangle

Syntax: **RUN GFX2([*path*,]"BOX"[,*xcor1,ycor1*], *xcor2,ycor2*)**

Function: Draws a rectangle. BOX defines its area with an imaginary diagonal line from the first set of coordinates to the second set of coordinates. BOX does not reset the draw pointer. The X coordinates are in the range 0-639. The Y coordinates are in the range 0-191.

Parameters:

path — The route to the window in which you want to draw a box.

xcor1,ycor1 — The beginning coordinates for the line that defines the rectangle to drawn. If you omit these coordinates, BOX uses the draw pointer position.

xcor2,cor2 — The ending coordinates for the line that defines the rectangular area to be drawn.

Examples:

```
RUN GFX2("BOX",200,100)

RUN GFX2("BOX",0,0,100,50)
```

Sample Program:

This procedure draws a series of progressively smaller boxes of different colors on a window screen. Then, it rapidly changes the colors of the boxes to produce a hypnotic effect.

```
PROCEDURE hypbox
DIM X,Y,X1,Y1,T,R,COLOR:INTEGER
DIM KEY:STRING[1]
KEY=""
X=18 \Y=6
Y1=185 \X1=621
RUN GFX2("CLEAR")
```

```
FOR T=0 TO 15
COLOR=T
RUN GFX2("COLOR",3)
RUN GFX2("BOX",X,Y,X1,Y1)
RUN GFX2("COLOR",COLOR)
RUN GFX2("FILL",X-1,Y-1)
X=X+18 \Y=Y+6
X1=X1-18 \Y1=Y1-6
NEXT T
WHILE KEY="" DO
RUN INKEY(KEY)
FOR T=1 TO 16
R=RND(65)
RUN GFX2("PALETTE",T,R)
NEXT T
ENDWHILE
RUN GFX2("DEFCOL")
END
```

CIRCLE Draw a circle

Syntax: **RUN GFX2([*path*,]"CIRCLE"[,*xcor,ycor*], *radius*)**

Function: Draws a circle with a specified radius. If you specify coordinates, CIRCLE uses them for the center point. Otherwise, CIRCLE locates the center of the circle at the current draw pointer position. See "The Graphics Cursor and the Draw Pointer" earlier in this section. Also see SETDPTR.

Parameters:

path The route to the window in which you want to draw a circle.

xcor,ycor The coordinates for the circle's center. The X coordinates are in the range 0-639. The Y coordinates are in the range 0-191.

radius The radius of the circle.

Examples:

```
RUN GFX2("CIRCLE",100)

RUN GFX2("CIRCLE",100,200,50)
```

Sample Program:

This procedure uses circles to produce a geometric design.

```
PROCEDURE ciraround
□DIM T,X,Y:INTEGER
□PRINT CHR$(12)
□RUN GFX2("COLOR",1,2)
□FOR T=1 TO 130
□X=150*SIN(T)+320
□Y=25*COS(T)+96
□RUN GFX2("CIRCLE",X,Y,100)
□NEXT T
□RUN GFX2("COLOR",3,2)
□FOR T=1 TO 45
□X=150*SIN(T)+320
□Y=25*COS(T)+96
□RUN GFX2("CIRCLE",X,Y,100)
□NEXT T
□END
```

CLEAR Clear the screen

Syntax: **RUN GFX2([*path*,]"CLEAR")**

Function: Clears the current working area of a window. CLEAR does not change the location of the draw pointer but does set the text cursor and graphics cursor location to the upper left corner of the window.

Parameters:

path The route to the window you want to clear.

Examples:

```
RUN GFX2("CLEAR")
```

COLOR Set screen colors

Syntax: **RUN GFX2([*path*,]"COLOR", *foreground*[,*background*][,*border*])**

Function: Changes any of the foreground, background, or the border colors. COLOR does not change the draw pointer position.

Parameters:

path — The route to the window in which you want to change one or more screen or text colors.

foreground — The register number for the foreground palette.

background — The register number for the background palette.

border — The register number for the border palette. Changing the border color for any window on a screen, changes the border color for all windows on the same screen.

Examples:

```
RUN GFX2("COLOR",1)
RUN GFX2("COLOR",1,2)
RUN GFX2("COLOR",1,2,1)
```

Sample Program:

This procedure fills a window screen with multicolored filled circles.

```
PROCEDURE bubbles
□DIM X,Y,W,Z,T:INTEGER
□Z=1
□RUN GFX2("COLOR",1,0,0)
□RUN GFX2("CLEAR")
□FOR T=1 TO 80
□X=RND(635)+4
□Y=RND(185)+5
□W=RND(50+5)
□Z=Z+1
□IF Z>3 THEN Z=1
□ENDIF
□RUN GFX2("CIRCLE",X,Y,W)
□RUN GFX2("COLOR",Z)
□RUN GFX2("FILL",X,Y)
□NEXT T
□RUN GFX2("COLOR",3,2,2)
□END
```

CRRTN Carriage return

Syntax: **RUN GFX2([*path*,]"CRRTN")**

Function: Causes BASIC09 to send a carriage return to a window. The cursor moves down one line and to the extreme left of the window.

Parameters:

path — The route to the window in which you want a carriage return.

Examples:

```
RUN GFX2("CRRTN")
```

CURDWN Cursor down

Syntax: **RUN GFX2([*path*,]"CURDWN")**

Function: Moves the cursor down one text line. The X-coordinate, or column position, remains the same.

Parameters:

path	The route to the window in which you want to move the cursor.

Examples:

```
RUN GFX2("CURDWN")
```

CURHOME **Cursor home**

Syntax: **RUN GFX2([*path*,]"CURHOME")**

Function: Moves the text cursor to the top left corner of the screen.

Parameters:

path	The route to the window where you want to reset the cursor

Examples:

```
RUN GFX2("CURHOME")
```

CURLFT Move cursor left

Syntax: **RUN GFX2([*path*,]"CURLFT")**

Function: Moves the cursor one character to the left.

Parameters:

path	The route to the window where you want to move the cursor.

Examples:

```
RUN GFX2("CURLFT")
```

CUROFF Turn off cursor

Syntax: **RUN GFX2([*path*,]"CUROFF")**

Function: Makes the cursor invisible.

Parameters:

path The route to the window in which you want to turn the cursor off.

Examples:

```
RUN GFX2("CUROFF")
```

CURON Turn on cursor

Syntax: **RUN GFX2([*path*,]"CURON")**

Function: Makes the text cursor visible.

Parameters:

path	The route to the window in which you want to turn the cursor on.

Examples:

```
RUN GFX2("CURON")
```

CURRGT Move cursor right

Syntax: **RUN GFX2("[*path*,]CURRGT")**

Function: Moves the cursor one character to the right.

Parameters:

path The route to the window in which you want to move the cursor.

Examples:

```
RUN GFX2("CURRGT")
```

CURUP **Move cursor up**

Syntax: **RUN GFX2([*path*,]"CURUP")**

Function: Moves the cursor up one line.

Parameters:

path The route to the window in which you want to move the cursor.

Examples:

```
RUN GFX2("CURUP")
```

CURXY Set cursor position

Syntax: **RUN GFX2([*path*,]"CURXY",*column*,*row*)**

Function: Moves the cursor to the specified column and row position. The column and row coordinates are relative to the window's current character width and depth.

Parameters:

path	The route to the window in which you want to move the cursor.
column	The column (horizontal) position for the cursor.
row	The row (vertical) position for the cursor.

Examples:

```
RUN GFX2("CURXY",10,10)
```

CWAREA Change working area

Syntax: **RUN GFX2([*path*,]"CWAREA",*xcor*,*ycor*,*sizex*, *sizey*)**

Function: Restricts output in the window to the specified area. The new area must be the same or smaller than the previous working area. When a window's working area is changed, OS-9 scales graphic and text coordinates and graphic images to the new proportions. Text characters remain the same size.

Parameters:

path — The route to the window in which you want to change the working area.

xcor,ycor — The beginning coordinates (the upper left corner) for the new working area, relative to the original window. The coordinates are based on the character column and row size of the original window.

sizex — Designates the number of columns in the new working area.

sizey — The number of lines available in the new working area.

Examples:

```
RUN GFX2("CWAREA",10,0,40,10)
```

Sample Program:

This procedure makes the working area in a window progressively smaller, filling each area with a different color. It then changes the areas' colors rapidly to produce a hypnotic effect.

```
PROCEDURE hypnobox
□DIM X,Y,X1,Y1,T,R,COLOR:INTEGER
□DIM KEY:STRING[1]
□KEY=""
□X=3 \Y=1
□X1=80-(X+X) \Y1=24-(Y+Y)
□FOR T=0 TO 10
□RUN GFX2("COLOR",3,T)
□RUN GFX2("CLEAR")
□RUN GFX2("CWAREA",X,Y,X1,Y1)
□X=X+3 \Y=Y+1
□X1=80-(X+X) \Y1=24-(Y+Y)
□NEXT T
□RUN GFX2("COLOR",3,2)
□WHILE KEY="" DO
□RUN INKEY(KEY)
□FOR T=1 TO 16
□R=RND(65)
□RUN GFX2("PALETTE",T,R)
□NEXT T
□ENDWHILE
□RUN GFX2("DEFCOL")
□RUN GFX2("CWAREA",0,0,80,24)
□END
```

DEFBUFF Define GET/PUT buffer

Syntax: **RUN GFX2("DEFBUFF",*group,buffer,size*)**

Function: Defines a buffer for GET/PUT operations.

When you define a buffer, you do so by group number and buffer number. Each group you define allocates eight kilobytes of memory. The system needs 30 bytes of the block for overhead, leaving 8162 bytes free. Within the group, you can allocate one or more buffers. Select a group number and a buffer number as indicated in the following "Parameters" section. Use these numbers in future references to the buffer.

A GET/PUT buffer remains allocated until you use the KILLBUFF function to remove it from your system's memory. For more information on Get/Put buffers, see KILLBUFF, PUT, GET, and GPLOAD.

Parameters:

group	A number you select in the range 1-199.
buffer	A number (in the range 1-255) that you assign to the buffer you create.
size	The size of the buffer, in the range of 1 to 8192 bytes, depending on available memory in its group.

Notes:

One method of selecting a group number is to use SYSCALL and the Get ID (103F 0C) system call to obtain your user's process ID number. Then, use this ID number as a group number. Using this system for all GET/PUT buffer operations, ensures against group number overlapping. See the SYSCALL command for more information.

Examples:

```
RUN GFX2("DEFBUFF",1,5,4000H0)
```

DEFCOL Set default colors

Syntax: **RUN GFX2([*path*,]"DEFCOL")**

Function: Sets the palette registers back to their default values. The type of monitor you have determines the actual hues. See "The Palette" and Table 9.7 earlier in this section.

Parameters:

path	The route to the window in which you want to restore the original palette registers.

Examples:

```
RUN GFX2("DEFCOL")
```

DELLIN Delete current line of text

Syntax: RUN GFX2([*path*,]"DELLIN")

Function: Deletes the line on which the cursor is resting and closes the space. DELLIN operates on both text and graphics screens.

Parameters:

path The route to the window in which you want to delete a line.

Examples:

```
RUN GFX2("DELLIN")
```

Sample Program:

This procedure draws a series of various colored concentric circles, then produces a lemon shape by removing slices of the circle with DELLIN.

```
PROCEDURE slice
□DIM X,Y,R,T,COLOR:INTEGER
□RUN GFX2("CLEAR")
□COLOR=0
□X=320
□Y=96
□FOR T=185 TO 10 STEP -10
□RUN GFX2("CIRCLE",X,Y,T)
□NEXT T
□FOR T=140 TO 320 STEP 10
□RUN GFX2("COLOR",COLOR)
□RUN GFX2("FILL",T,96)
□COLOR=COLOR+1
□NEXT T
□RUN GFX2("CURXY",0,8)
□FOR T=1 TO 8
□RUN GFX2("DELLIN")
□NEXT T
□RUN GFX2("COLOR",3,2)
□END
```

DRAW **Draw a polyline figure**

Syntax: **RUN GFX2([*path*,]"DRAW",*option list*)**

Function: Draws in the directions specified, and for the distances specified, in an option list. The option list is a string of characters and numbers. You can separate options with spaces or commas. You must include commas between the two coordinates for the B and U options.

Parameters:

path	The route to the window in which you want to draw.
option list	A string consisting of one or more of the following options:

Options:

N*num*	draws north (up) *num* units.
S*num*	draws south (down) *num* units.
E*num*	draws east (right) *num* units.
W*num*	draws west (left) *num* units.
NE*num*	draws northeast (up and right) *num* units.
NW*num*	draws northwest (up and left) *num* units.
SE*num*	draws southeast (down and right) *num* units.
SW*num*	draws southwest (down and left) *num* units.
A*val*	rotates the draw axis. Possible values are: 0 = normal 1 = 90 degrees 2 = 180 degrees 3 = 270 degrees
U*xcor,ycor*	draws a relative vector to the specified coordinates. *Xcor* and *ycor* are relative to the current draw pointer position. The draw pointer location does not change. *Xcor* and *ycor* must be separated by a comma.

B*xcor,ycor* produces a blank line (moves the cursor but does not draw). The *xcor* and *ycor* coordinates are relative to the current draw pointer location. If you specify relative coordinates located offscreen, you cannot see subsequent lines.

Examples:

```
RUN GFX2("DRAW","N10,E10,S10,W10")
```

Sample Program:

```
PROCEDURE drawing
□DIM T,X,Y,COLOR:INTEGER
□COLOR=0
□RUN GFX2("CLEAR")
□FOR T=1 TO 96 STEP 6
□RUN GFX2("SETDPTR",320,96)
□FOR Y=0 TO 3
□COLOR=MOD(Y,2)
□RUN GFX2("COLOR",COLOR)
□FOR X=1 TO 4
□READ DR$
□DR$="A"+STR$(Y)+DR$+STR$(T)
□RUN GFX2("DRAW",DR$)
□NEXT X
□NEXT Y
□RESTORE
□NEXT T
□RUN GFX2("COLOR",3)
□END
□DATA "N","E","S","W"
```

DWEND Device window end

Syntax: RUN GFX2([*path*,]"DWEND")

Function: Deallocates the device window you initialized with DWSET and INIZ. If the window deallocated is the last device window on the screen, BASIC09 returns the screen memory to the system. DWEND automatically positions you in the next device window, a result similar to pressing CLEAR. You can use this function with DWSET to redefine a device window to a different type.

Parameters:

path — The path number of the window you wish to end. Path can be a constant or variable.

Examples:

```
RUN GFX2("DWEND")
RUN GFX2(PATH,"DWEND")
RUN GFX2(3,"DWEND")
```

Sample Program:

From /TERM, this procedure temporarily opens a path to Window 3, displays the new window, draws a design, then returns to the /TERM screen and closes the path.

```
PROCEDURE decorate
□DIM PATH,T,Y:INTEGER
□OPEN #PATH,"/W3":WRITE
□RUN GFX2(PATH,"DWSET",7,0,0,80,24,3,2,2)
□RUN GFX2(PATH,"SELECT")
□Y=1
□RUN GFX2(PATH,"COLOR",3,2)
□FOR T=1 TO 185 STEP 3
□Y=Y+1
□RUN GFX2(PATH,"ELLIPSE",320,96,T,Y)
□NEXT T
□RUN GFX2(PATH,"COLOR",1,2)
□FOR T=185 TO 1 STEP -6
```

```
□RUN GFX2(PATH,"ELLIPSE",320,96,T,Y)
□IF INT(T/3)=T/3 THEN
□Y=Y+1
□ENDIF
□NEXT T
□RUN GFX2(1,"SELECT")
□RUN GFX2(PATH,"DWEND")
□CLOSE #PATH
□END
```

DWPROTSW Device window protect switch

Syntax: **RUN GFX2([*path*,]"DWPROTSW","*switch*")**

Function: Lets you *unprotect* one device window and set other device windows on top of it.

OS-9 on the Color Computer 3 normally uses a protected windowing system that does not allow window devices to overlap. Removing the window protection with DWPROTSW lets one device window exist on the same screen area as another window device. Because this might destroy the contents of an unprotected window, you need to use care with this function.

Parameters:

path The route to the window you want to unprotect.

switch Either OFF to turn off protection, or ON to turn on protection. The default is ON.

Examples:

```
RUN GFX2("DWPROTSW",OFF)
```

DWSET Device window set

Syntax: **RUN GFX2([*path*,]"DWSET",*format,xcor,ycor, width,length,foreground,background,border*)**

Function: Defines a device window. Normally, you first open a path to a window, then use DWSET to set the window format, location, size, and colors.

Parameters:

path	The route to the window you are defining.
format	The code for the type of screen you want to establish. See Table 9.6 at the beginning of this section for the formats available.
xcor,ycor	The coordinates (character column and row) of the upper left corner of the screen you want to create.
width	The width (in characters) of the new window.
length	The depth (in lines) of the new window.
foreground	The code for the window's foreground color.
background	The code for the window's background color.
border	The code for the window's border color.

Examples:

```
RUN GFX2("DWSET",06,50,100,50,10,20,12,9)
```

Sample Program:

This procedure opens a path to Window 3, uses DWSET to define the new window, displays the new window, and draws a graphic *lemon* shape. It then uses SELECT to return to the /TERM window or screen, deallocates Window 3, and closes the path.

```
PROCEDURE lemon
□DIM PATH,T,X,Y:INTEGER
□OPEN #PATH,"/W3":WRITE
□RUN GFX2(PATH,"DWSET",7,0,0,80,24,3,2,2)
□RUN GFX2(PATH,"SELECT")
□Y=1
□RUN GFX2(PATH,"COLOR",0,2)
□FOR T=1 TO 185 STEP 3
□Y=Y+1
□RUN GFX2(PATH,"ELLIPSE",320,96,T,Y)
□NEXT T
□X=T
□RUN GFX2(PATH,"COLOR",3,2)
□FOR T=62 TO 1 STEP -3
□RUN GFX2(PATH,"ELLIPSE",320,96,X,T)
□IF INT(T/3)=T/3 THEN
□X=X+1
□ENDIF
□NEXT T
□RUN GFX2(1,"SELECT")
□RUN GFX2(PATH,"DWEND")
□CLOSE #PATH
□END
```

ELLIPSE Draw an ellipse

Syntax: **RUN GFX2([*path*,]"ELLIPSE"[,*xcor*,*ycor*], *xrad*,*yrad*)**

Function: Draws an ellipse with the center at the current draw pointer position or at the specified X,Y coordinates. The X coordinates are in the range 0-639. The Y coordinates are in the range 0-191.

Parameters:

path	The route to the window in which you want to draw.
xcor,*ycor*	The coordinates for the ellipse's center. If you omit these coordinates, ELLIPSE uses the current draw pointer position.
xrad,*yrad*	The radii of the ellipse's length and height.

Examples:

```
RUN GFX2("ELLIPSE",100,50)

RUN GFX2("ELLIPSE",100,125,100,10)
```

Sample Program:

This program uses ELLIPSE to draw a graphic design shaped like a Christmas tree decoration.

```
PROCEDURE xbulb
□DIM T,Y:INTEGER
□Y=1
□RUN GFX2("COLOR",3,2)
□RUN GFX2("CLEAR")
□FOR T=1 TO 180 STEP 3
□Y=Y+1
□RUN GFX2("ELLIPSE",320,96,T,Y)
□NEXT T
□RUN GFX2("COLOR",1,2)
□FOR T=180 TO 1 STEP -6
```

```
□RUN GFX2("ELLIPSE",320,96,T,Y)
□IF INT(T/3)=T/3 THEN
□Y=Y+1
□ENDIF
□NEXT T
□RUN GFX2("COLOR",3,2)
□END
```

EREOLINE Erase to end of line

Syntax: RUN GFX2([*path*,]"EREOLINE")

Function: Deletes the portion of the current line from the cursor to the right side of the window.

Parameters:

path The route to the window in which you want to erase a portion of a line.

Examples:

```
RUN GFX2("EREOLINE")
```

Sample Program:

This procedure uses EREOLINE to produce a series of steps down the screen.

```
PROCEDURE steps
□DIM T,J,K:INTEGER
□RUN GFX2("COLOR",2,3)
□RUN GFX2("CLEAR")
□RUN GFX2("COLOR",3,2)
□FOR T=0 TO 22
□J=T*3
□RUN GFX2("CURXY",J,T)
□RUN GFX2("EREOLINE")
□NEXT T
```

EREOWNDW **Erase to end of window**

Syntax: **RUN GFX2([*path*,]"EREOWNDW")**

Function: Deletes all the lines in a window from the line on which the cursor is positioned to the bottom of the window.

Parameters:

path The route to the window in which you want to delete screen contents.

Examples:

```
RUN GFX2("EREOWNDW")
```

ERLINE Delete current line of text

Syntax: **RUN GFX2([*path*,]"ERLINE")**

Function: Deletes the current line (on which the cursor is resting) from the window but does not close the space.

Parameters:

path	The route to the window in which you want to remove the contents of a screen line.

Examples:

```
RUN GFX2("ERLINE")
```

Sample Program:

This procedure draws a bull's-eye design, then slices it with the ERLINE function.

```
PROCEDURE cut
□DIM X,Y,R,T,COLOR:INTEGER
□COLOR=0
□X=320
□Y=96
□RUN GFX2("CLEAR")
□COLOR=0
□FOR T=185 TO 10 STEP -10
□RUN GFX2("CIRCLE",X,Y,T)
□NEXT T
□FOR T=140 TO 320 STEP 10
□RUN GFX2("COLOR",COLOR)
□RUN GFX2("FILL",T,96)
□COLOR=COLOR+1
□NEXT T
□FOR T=2 TO 22 STEP 2
□RUN GFX2("CURXY",0,T)
□RUN GFX2("ERLINE")
□NEXT T
□RUN GFX2("COLOR",3,2)
□END
```

FILL Fill (paint) window

Syntax: **RUN GFX2([*path*,]"FILL",[*xcor,ycor*])**

Function: Paints an area with the current foreground color. Paint fills the portion of the window that is the same color as the pixel under the draw pointer.

Parameters:

path	The route to the window in which you want to use the FILL function.
xcor,ycor	Are optional X- and Y-coordinates to reposition the draw pointer before FILL begins. If you omit these coordinates, BASIC09 uses the current draw position.

Examples:

```
RUN GFX2("FILL",100,100)
```

Sample Program:

This procedure draws and fills 100 boxes on a window.

```
PROCEDURE colorbox
□DIM A,B,C,D,T,COLOR:INTEGER
□COLOR=0
□RUN GFX2("CLEAR")
□FOR T=1 TO 100
□A=RND(560)
□B=RND(151)
□C=A+RND(80)
□D=B+RND(40)
□COLOR=COLOR+1
□RUN GFX2("COLOR",COLOR)
□RUN GFX2("BOX",A,B,C,D)
□RUN GFX2("FILL",A+1,B+1)
□NEXT T
```

FONT Define font buffer

Syntax: **RUN GFX2([*path*,]"FONT",*group*,*buffer*)**

Function: Defines a buffer from which BASIC09 gets the character font (style) for the current screen. Use the text/cursor handling functions referenced in this section with the font you load. When you merge the Stdfonts file in your SYS directory with a graphics window, you have the choice of three fonts from Buffers 1, 2, and 3, located in Group 200. You can also create your own fonts. FONT works only on graphics screen. See "Using Fonts" earlier in this chapter.

You must load the font you want to use into the defined buffer before using FONT.

Parameters:

path	The route to the window in which you want to use an alternate font.
group	The group number of the buffer containing the font to use.
buffer	The number of the buffer containing the font to use.

Examples:

```
RUN GFX2("FONT",200,2)
```

GCSET **Set graphics cursor**

Syntax: **RUN GFX2("GCSET",*group,buffer*)**

Function: Defines a buffer from which BASIC09 gets the graphics cursor. This lets you define your own cursor for graphics operations. To turn the graphics cursor off, use a group Number 0. You must execute this command to display a graphics cursor. Before using GCSET, you must merge the Stdcur file in the SYS directory to the window.

Parameters:

group	The group number of the buffer containing the cursor image to use. See *OS-9 Windowing System* for information on the group to use.
buffer	The number of the buffer that contains the cursor image to use. See *OS-9 Windowing System* for information on the buffer to use.

Examples:

```
RUN GFX2("GCSET",1,5)
```

GET Get a block from the window

Syntax: **RUN GFX2([*path*,]"GET",*group*,*buffer*,*xcor*, *ycor*,*xsize*,*ysize*)**

Function: Saves a window area Get/Put buffer. Use PUT to replace the image to the window. If you did not previously define the buffer, BASIC09 creates it. If you store the window data in a predefined buffer, the data must be the same size or smaller than the buffer. If not, BASIC09 truncates the data to the size of the buffer. (Also see PUT and DEFBUFF.)

Parameters:

path	The route to the window where you want to save an image.
group	The group number of the Get buffer (1-199).
buffer	The Get buffer number (1-255).
xcor,ycor	The X- and Y-coordinates of the upper left corner of the window image to save. The X-coordinates are in the range 0-639. The Y-coordinates are in the range 0-191.
xsize	The horizontal size of the window section to save.
ysize	The vertical size of the window section to save.

Examples:

```
RUN GFX2("GET",1,5,0,0,10,15)
```

Sample Program:

This procedure draws a character, loads it into a buffer, then repeatedly replaces the character to the window screen using PUT. Each new image erases the previous image, giving an impression of animation.

```
PROCEDURE puttdown
□DIM T,J:INTEGER
□RUN GFX2("CLEAR")
□RUN GFX2("ELLIPSE",320,96,12,4)
□RUN GFX2("CIRCLE",320,90,5)
□RUN GFX2("COLOR",1)
□RUN GFX2("FILL",320,96)
□RUN GFX2("COLOR",3)
□RUN GFX2("FILL",320,90)
□RUN GFX2("BAR",305,100,335,104)
□RUN GFX2("GET",1,1,288,85,50,23)
□RUN GFX2("GET",1,2,1,1,50,23)
□RUN GFX2("PUT",1,2,288,85)
□J=10
□FOR T=20 TO 559 STEP 6
□J=J+2
□RUN GFX2("PUT",1,1,T,J)
□NEXT T
□RUN GFX2("KILLBUFF",1,1)
□RUN GFX2("CURON")
□END
```

GPLOAD Load data into Get/Put buffer

Syntax: **RUN GFX2("GPLOAD",*group,buffer,format,xdim,ydim,size*)**

Function: Loads a buffer with image data that PUTBLK can use for window displays. If the Get/Put buffer is not created, BASIC09 creates it. If it is defined, the size of the data should not be larger than the buffer.

Parameters:

group	The group number you select, in the range 1-199, to let you group buffers.
buffer	A number in the range 1-255 that you assign to the buffer you create.
format	The type code of the screen format. (See Table 9.4.)
xdim	The X (horizontal) dimension of the stored block.
ydim	The Y (vertical) dimension of the stored block.
size	The size of the buffer in bytes. A buffer size can be in the range of 1 to 8 kilobytes, depending on available memory.

Examples:

```
RUN GFX2("DEFBUFF",1,5,06,100,50,5000)
```

INSLIN **Insert line**

Syntax: **RUN GFX2([*path*,]"INSLIN")**

Function: Moves the window lines at and below the cursor down one line.

Parameters:

path	The route to the window in which you want a blank line.

Examples:

```
RUN GFX2("INSLIN")
```

Sample Program:

This procedure draws a round face on the screen, then uses INSLIN and DELLIN to make a mouth appear to move.

```
PROCEDURE chomp
□DIM X,Y,T:INTEGER
□DIM RESPONSE:STRING[1]
□RESPONSE=""
□RUN GFX2("CLEAR")
□RUN GFX2("CIRCLE",320,96,80)
□RUN GFX2("COLOR",0,2)
□RUN GFX2("FILL",320,96)
□RUN GFX2("COLOR",2)
□RUN GFX2("CIRCLE",285,80,12)
□RUN GFX2("CIRCLE",355,80,12)
□RUN GFX2("FILL",285,80)
□RUN GFX2("FILL",355,80)
□RUN GFX2("CIRCLE",315,96,3)
□RUN GFX2("CIRCLE",325,96,3)
□RUN GFX2("ARC",320,92,14,3,3,1,1,1)
□RUN GFX2("COLOR",3,2)
□RUN GFX2("CIRCLE",289,77,3)
□RUN GFX2("CIRCLE",359,77,3)
□RUN GFX2("CURXY",0,14)
□REPEAT
```

```
□RUN GFX2("INSLIN")
□FOR X=1 TO 100
□NEXT X
□RUN GFX2("DELLIN")
□RUN INKEY(RESPONSE)
□UNTIL RESPONSE>""
□END
```

KILLBUFF Deallocate Get/Put buffer

Syntax: RUN GFX2("KILLBUFF",*group,buffer*)

Deallocates the indicated Get/Put buffer. You select group and buffer numbers when you define a buffer or when you load or get a window image. For more information on Get/Put buffers, see DEFBUFF, PUT, GET, and GPLOAD.

Parameters:

group The group number of the buffer you want to deallocate, in the range 1-199. Buffer Group Numbers 0 and 200-255 are reserved for OS-9 system use.

buffer The number of the buffer to deallocate, in the range 1-255.

Examples:

```
RUN GFX2("KILLBUFF",1,5)
```

Sample Program:

This procedure draws a figure on a window screen, loads it into a buffer, then repeatedly places it in new locations on the screen. Each new PUT erases the previous image.

```
PROCEDURE putdown
□DIM X,Y,T,J:INTEGER
□RUN GFX2("CUROFF")
□RUN GFX2("CLEAR")
□RUN GFX2("ELLIPSE",320,96,12,4)
□RUN GFX2("CIRCLE",320,90,5)
□RUN GFX2("COLOR",1)
□RUN GFX2("FILL",320,96)
□RUN GFX2("COLOR",3)
□RUN GFX2("FILL",320,90)
□RUN GFX2("BAR",305,100,335,104)
□RUN GFX2("GET",1,1,288,85,50,23)
□RUN GFX2("GET",1,2,1,1,50,23)
□RUN GFX2("PUT",1,2,288,85)
```

```
□J=10
□FOR T=20 TO 559 STEP 6
□J=J+2
□RUN GFX2("PUT",1,1,T,J)
□NEXT T
□RUN GFX2("KILLBUFF",1,1)
□RUN GFX2("CURON")
□END
```

LINE **Draw a line**

Syntax: **RUN GFX2([*path*,]"LINE"[,*xcor1*,*ycor1*],*xcor2*, *ycor2*)**

Function: Draws a line in one of the following ways:

- From the current draw pointer to the specified X- and Y-coordinates.
- From the specified beginning X- and Y-coordinates to the specified ending X- and Y-coordinates.

Parameters:

path	The route to the window in which you want to draw a line.
xcor1,ycor1	The optional beginning X- and Y-coordinates for the line.
xcor2,ycor2	The ending X- and Y-coordinates for the line.

Examples:

```
RUN GFX2("LINE",192,128)

RUN GFX2("LINE",0,0,192,128)
```

Sample Program:

This procedure draws a sine wave of vertical lines across a window.

```
PROCEDURE waves
□DIM A,X,Y,Z:INTEGER
□CALC=0
□A=100
□RUN GFX2("CLEAR")
□RUN GFX2("COLOR",3,2)
□FOR X=0 TO 638 STEP 1
□CALC=CALC+.05
□Y=A-SIN(CALC)*15
□Z=Y+25
```

```
□RUN GFX2("LINE",X,Y,X,Z)
□NEXT X
□END
```

LOGIC **Perform logic function**

Syntax: **RUN GFX2("LOGIC","*function*")**

Function: Causes BASIC09 to perform the specified logic function on all data bits used by subsequent drawing functions. Once set, the logic function remains in effect until you turn LOGIC off.

Parameters:

function can be one of the following logical functions:

OFF	— no logic
AND	— performs AND logic
OR	— performs OR logic
XOR	— performs XOR logic

Examples:

```
RUN GFX2("LOGIC","AND")

RUN GFX2("LOGIC","XOR")
```

Sample Program:

This procedure uses LOGIC to draw a horizontal bar across a background of multicolored vertical bars. Using XOR logic, the procedure causes the horizontal bar to change the color of each vertical bar.

```
PROCEDURE logic
□DIM A,Z,T,X,Y,COLOR:INTEGER
□RUN GFX2("LOGIC","OFF")
□RUN GFX2("CLEAR")
□COLOR=0
□FOR T=0 TO 619 STEP 20
□COLOR=COLOR+1
□RUN GFX2("COLOR",COLOR)
□RUN GFX2("BAR",T,0,T+20,190)
□NEXT T
□RUN GFX2("COLOR",3,2)
□RUN GFX2("LOGIC","XOR")
```

```
□FOR T=1 TO 10
□RUN GFX2("BAR",0,80,639,112)
□NEXT T
□RUN GFX2("LOGIC","OFF")
□END
```

OWSET **Establish an overlay window**

Syntax: **RUN GFX2([*path,*]"OWSET",*save switch,xpos, ypos,xsize,ysize,foreground,background*)**

Function: Creates an overlay window on a previously existing device window. Reconfigures the current device window paths to use a new area of the screen as the current device window.

Parameters:

path The route to the window in which you want to set an overlay.

save switch Either 0 or 1. A value of 0 tells BASIC09 not to save the overlaid area. A value of 1 tells BASIC09 to save the overlaid area and restore it when the new window closes.

xpos The character column in which to start the new window (upper left corner).

ypos The character row in which to start the new window (upper left corner).

xsize The width of the new window in characters.

ysize The depth of the new window in rows.

foreground The foreground color of the new window.

background The background color of the new window.

Examples:

```
RUN GFX2("OWSET",00,44,10,32,8,00,06)
```

Sample Program:

This procedure creates six progressively smaller overlay windows, labeling each. It then waits for you to press a key, after which it erases all the windows and leaves the original window intact.

```
PROCEDURE overwin
□DIM X,Y,X1,Y1,T,J,B,L,PLACE:INTEGER
□DIM RESPONSE:STRING[1]
□X=0 \Y=0
□X1=80 \Y1=24
□PLACE=33
□FOR T=1 TO 6
□IF T=2 OR T=6 THEN
□B=3
□ELSE B=2
□ENDIF
□RUN GFX2("OWSET",1,X,Y,X1,Y1,B,T)
□X=X+6 \Y=Y+2
□X1=X1-12 \Y1=Y1-4
□FOR J=1 TO 5
□PRINT TAB(PLACE); "Overlay Screen "; T
□NEXT J
□PLACE=PLACE-6
□NEXT T
□PRINT "Press A Key...";
□GET #1,RESPONSE
□FOR T=1 TO 6
□RUN GFX2("OWEND")
□NEXT T
□END
```

PALETTE Set color for palette registers

Syntax: **RUN GFX2([*path,*]"PALETTE",*register,color*)**

Function: Sets palette colors. Lets you *install* any of the Color Computer's 64 colors in the palette for use with text and graphics.

Parameters:

path	The route to the window where you want to change palette colors.
register	The number of the register in which you want to install a new color.
color	The code of the new color you want to install.

Examples:

```
RUN GFX2("PALETTE",13,32)
```

Sample Program:

This procedure draws a series of bars and circles, then repeatedly changes their colors using PALETTE.

```
PROCEDURE palette
□DIM T,K,J,X,Y,COLOR:INTEGER
□DIM RESPONSE:STRING[1]
□RUN GFX2("COLOR",3,2,2)
□COLOR=0
□RUN GFX2("CLEAR")
□RUN GFX2("CUROFF")
□FOR Y=0 TO 23 STEP 3
□RUN GFX2("COLOR",COLOR)
□RUN GFX2("BAR",0,Y,639,Y+3)
□COLOR=COLOR+1
□IF COLOR=2 THEN
□COLOR=COLOR+1
□ENDIF
□NEXT Y
□FOR Y=164 TO 185 STEP 3
```

```
□RUN GFX2("COLOR",COLOR)
□RUN GFX2("BAR",0,Y,639,Y+3)
□COLOR=COLOR+1
□NEXT Y
□COLOR=0
□FOR K=45 TO 170 STEP 48
□FOR T=100 TO 580 STEP 100
□RUN GFX2("COLOR",3)
□RUN GFX2("CIRCLE",T,K,30)
□RUN GFX2("COLOR",COLOR)
□RUN GFX2("FILL",T,K)
□COLOR=COLOR+1
□IF COLOR=2 THEN
□COLOR=COLOR+1
□ENDIF
□NEXT T
□NEXT K
□REPEAT
□X=RND(63)
□REPEAT
□Y=RND(16)+1
□UNTIL Y<>2
□RUN GFX2("PALETTE",Y,X)
□RUN INKEY(RESPONSE)
□UNTIL RESPONSE>""
□RUN GFX2("DEFCOL")
□RUN GFX2("CURON")
□END
```

PATTERN Select pattern buffer

Syntax: **RUN GFX2([*path*,]"PATTERN",*group*,*buffer*)**

Function: Selects the contents of a preloaded Get/Put buffer as a pattern for graphics functions. Although PATTERN can use a buffer of any size, it uses a specific number of bytes, depending on the screen format in use:

Color Mode	Pattern Array Size	Bits Per Pel
02	4 bytes x 8 bytes = 32 bytes	1
04	8 bytes x 8 bytes = 64 bytes	2
16	16 bytes x 8 bytes = 128 bytes	4

The pattern array is a 32 x 8 pel representation of graphics memory. It takes the current color mode into consideration to define the number of bits per pel and pels per byte. If the buffer is larger than the number of bytes required, PATTERN ignores the extra bytes. BASIC09 uses the selected pattern with all draw commands until you change the pattern or turn off the pattern function by specifying a group and buffer number of 0.

Parameters:

path — The route to the window in which you want to use a new graphics pattern.

group — The group number of the buffer you want to use for a graphics pattern.

buffer — The buffer number that you want to use for a graphics pattern.

Examples:

```
RUN GFX2("PATTERN",1,3)
```

Sample Program:

This procedure loads the current window data at location 0,0 into a buffer to use as a draw pattern. It then draws a circle and fills the circle with the pattern in the buffer.

```
PROCEDURE pattern
□DIM X,Y,T:INTEGER
□RUN GFX2("GET",1,1,0,0,5,5)
□RUN GFX2("COLOR",4)
□RUN GFX2("CLEAR")
□RUN GFX2("CIRCLE",320,96,100)
□RUN GFX2("FILL",320,96)
□RUN GFX2("PATTERN",1,1)
□RUN GFX2("COLOR",3)
□RUN GFX2("FILL",320,96)
□RUN GFX2("PATTERN",0,0)
□END
```

POINT Mark a point

Syntax: **RUN GFX2([*path*,]"POINT"[,*xcor,ycor*])**

Function: Sets the pixel at the current draw pointer position or at the specified coordinates to the current foreground color. If you do not specify coordinates, POINT sets the pixel at the draw pointer.

Parameters:

path	The route to the window in which you want to turn on the specified pixels.
xcor,ycor	Optional coordinates for the POINT function. The X-coordinates are in the range 0-639. The Y-coordinates are in the range 0-191.

Examples:

```
RUN GFX2("POINT")

RUN GFX2("POINT",192,128)
```

Sample Program:

This procedure uses POINT to produce a *swirl* design on a window screen.

```
PROCEDURE point
□BASE 0
□DIM X(20),Y(20):INTEGER
□DIM T,R,J,K:INTEGER
□J=0
□K=0
□RUN GFX2("CUROFF")
□RUN GFX2("CLEAR")
□FOR T=1 TO 288 STEP 3
□J=J+1
□FOR R=0 TO 11
□X(R)=INT(T*SIN(30*R+K))+320
□Y(R)=INT(J*COS(30*R+K))+96
□RUN GFX2("POINT",X(R),Y(R))
```

```
 K=K+1
 NEXT R
 NEXT T
 RUN GFX2("CURON")
 END
```

PROPSW Proportional space switch

Syntax: **RUN GFX2([*path*,]"PROPSW","*switch*")**

Function: Enables or disables the automatic proportional spacing of characters on graphic screens.

Parameters:

path	The route to the window in which you want to use proportional character spacing.
switch	Either OFF to turn proportional spacing off, or ON to turn proportional spacing on. The default setting of the switch is OFF.

Examples:

```
RUN GFX2("PROPSW","ON")
```

Sample Program:

This procedure produces a demonstration of the BASIC09 proportional spacing function.

```
PROCEDURE proport
□DIM LINE:STRING
□DIM LETTER:STRING[1]
□DIM T,J,K,FLAG:INTEGER
□RUN GFX2("CLEAR")
□FLAG=1
□FOR T=1 TO 12
□READ LINE
□FOR J=1 TO LEN(LINE)
□LETTER=MID$(LINE,J,1)
□IF LETTER<>"!" AND LETTER<>"#" THEN
□PRINT LETTER;
□ENDIF
□IF LETTER="!" THEN
□FLAG=FLAG*-1
□IF FLAG>0 THEN
□RUN GFX2("PROPSW","OFF")
```

```
ELSE
RUN GFX2("PROPSW","ON")
ENDIF
ENDIF
IF LETTER="#" THEN
PRINT CHR$(34);
ENDIF
NEXT J
PRINT
NEXT T
PRINT \ PRINT
END
DATA "This is a demonstration of"
DATA "!Proportional Spacing! using"
DATA "BASIC09's GFX2 module."
DATA ""
DATA "!The quick brown fox jumped...!"
DATA "The quick brown fox jumped..."
DATA ""
DATA "Use the command"
DATA "!RUN GFX2(#PROPSW#,#ON#)!"
DATA "to turn proportional spacing on."
DATA "Use !RUN GFX2(#PROPSW#,#OFF#)!"
DATA "to turn proportional spacing off"
```

PUT Put a saved data block on the window

Syntax: **RUN GFX2([*path*,]"PUT",*group*,*buffer*, *xcor*,*ycor*)**

Function: Places the image in the specified Get/Put buffer on the window. PUT requires only the group and buffer numbers and the window coordinates for the upper left corner of the image. The GET function saves the dimensions of the block in the buffer. PUT automatically handles window format conversion.

Parameters:

path	The route to the window where you want to place a pre-saved image.
group	The group number of the buffer in which to save the window data.
buffer	The buffer number in which to save the window data.
xcor,ycor	The X- and Y-coordinates of the upper left corner of the window position. The X-coordinates are in the range 0-639. The Y-coordinates are in the range 0-191.

Examples:

```
RUN GFX2("PUT",1,5,100,50)
```

Sample Program:

This procedure draws a character, loads it into a buffer, then repeatedly replaces the character to the window screen using PUT. Each new image erases the previous image, giving an impression of animation.

```
PROCEDURE putdown
  DIM X,Y,T,J:INTEGER
  RUN GFX2("CUROFF")
  RUN GFX2("CLEAR")
  RUN GFX2("ELLIPSE",320,96,12,4)
  RUN GFX2("CIRCLE",320,90,5)
  RUN GFX2("COLOR",1)
  RUN GFX2("FILL",320,96)
  RUN GFX2("COLOR",3)
  RUN GFX2("FILL",320,90)
  RUN GFX2("BAR",305,100,335,104)
  RUN GFX2("GET",1,1,288,85,50,23)
  RUN GFX2("GET",1,2,1,1,50,23)
  RUN GFX2("PUT",1,2,288,85)
  J=10
  FOR T=20 TO 559 STEP 6
  J=J+2
  RUN GFX2("PUT",1,1,T,J)
  NEXT T
  RUN GFX2("KILLBUFF",1,1)
  RUN GFX2("CURON")
  END
```

PUTGC **Put graphics cursor**

Syntax: **RUN GFX2([*path*,]"PUTGC",*xcor,ycor*)**

Function: Places and displays the graphics cursor at the specified location. Use screen relative coordinates for this function, not window relative coordinates. The horizontal range is 0-639. The vertical range is 0-191.

Parameters:

path	The route to the window where you want to display a graphics cursor.
xcor,ycor	The screen coordinates for the cursor location. The X coordinates are in the range 0-639. The Y coordinates are in the range 0-191.

Examples:

```
RUN GFX2("PUTGC",100,5)
```

Sample Program:

This procedure displays the available graphic cursors stored in group 202. Before this procedure can work, you must merge the Stdptrs file in the SYS directory of your system disk with the window you are using. For instance, if your system diskette is in Drive /D0, merge Stdptrs with Window 1, by typing:

```
merge /d0/sys/stdptrs > /w1 [ENTER]
```

```
PROCEDURE viewcur
□DIM T,Z:INTEGER
□RUN GFX2("CLEAR")
□FOR T=1 TO 7
□RUN GFX2("GCSET",202,T)
□RUN GFX2("PUTGC",320,96)
□FOR Z=1 TO 6000
□NEXT Z
□NEXT T
□RUN GFX2("GCSET",0,0)
□END
```

REVON **Reverse video on**
REVOFF **Reverse video off**

Syntax: **RUN GFX2([*path*,]"REVON")**
RUN GFX2([*path*,]"REVOFF")

Function: Enables or disables reverse video characters. Once set, reverse video remains in effect until you execute the reverse video off function.

Parameters:

path	The route to the window in which you want to display reverse characters.

Examples:

```
RUN GFX2("REVON")
RUN GFX2("REVOFF")
```

SCALESW **Enable/disable scaling**

Syntax: **RUN GFX2([*path*,]"SCALESW","*switch*")**

Function: Enables or disables scaling when drawing on variously formatted windows. Scaling in windows is normally on. If scaling is off, coordinates are relative to the window origin coordinates. Scaling does not affect text.

Parameters:

path	The route to the window where you want to turn scaling off or on.
switch	Either OFF (disable scaling) or ON (enable scaling).

Examples:

```
RUN GFX2("SCALESW","OFF")
```

Sample Program:

This procedure runs a routine of drawing a design in overlay windows twice. The routine runs once with scaling off and once with scaling on. After the first routine pauses, press the space bar to see the second demonstration.

```
PROCEDURE scale
□DIM X,Y,X1,Y1,T,B,J,R,W,Z:INTEGER
□DIM RESPONSE:STRING[1]
□RUN GFX2("CLEAR")
□FOR J=1 TO 2
□IF J=1 THEN
□RUN GFX2("SCALESW","OFF")
□ELSE
□RUN GFX2("SCALESW","ON")
□ENDIF
□X=0 \Y=0 \X1=80 \Y1=24
□FOR T=1 TO 4
□IF T=2 OR T=6 THEN
□B=3
```

```
□ELSE B=2
□ENDIF
□RUN GFX2("OWSET",1,X,Y,X1,Y1,B,T)
□FOR R=1 TO 35
□W=40*SIN(R)+170
□Z=25*COS(R)+45
□RUN GFX2("CIRCLE",W,Z,30)
□NEXT R
□X=X+6 \Y=Y+2 \X1=X1-12 \Y1=Y1-4
□NEXT T
□PRINT "Press A Key...";
□GET #1,RESPONSE
□FOR T=1 TO 4
□RUN GFX2("OWEND")
□NEXT T
□NEXT J
□END
```

SELECT **Select next window**

Syntax: **RUN GFX2([*path*],"SELECT")**

Function: SELECT causes a window to display if the procedure is operating in the active window. If the procedure is not in the active window, the newly selected window displays when you press CLEAR. If you do not specify a path, BASIC09 selects the device using the standard input, standard output, and standard error paths, Paths 0, 1, and 2.

Parameters:

path The path to the window to select.

Examples:

```
RUN GFX2("SELECT")

RUN GFX2(1,"SELECT")

RUN GFX2(PATH,"SELECT")
```

Sample Program:

From /TERM, this procedure temporarily opens a path to Window 3, creates the window format, and uses SELECT to display the new window. It draws a design, then returns to the /TERM screen and closes the path.

```
PROCEDURE design
□DIM PATH,T,Y:INTEGER
□OPEN #PATH,"/W3":WRITE
□RUN GFX2(PATH,"DWSET",5,0,0,80,24,3,2,2)
□RUN GFX2(PATH,"SELECT")
□Y=1
□FOR T=1 TO 200 STEP 3
□Y=Y+1
□RUN GFX2(PATH,"ELLIPSE",320,96,T,Y)
□NEXT T
□RUN GFX2(PATH,"COLOR",1,2)
□FOR T=200 TO 1 STEP -6
□RUN GFX2(PATH,"ELLIPSE",320,96,T,Y)
```

```
□IF INT(T/3)=T/3 THEN
□Y=Y+1
□ENDIF
□NEXT T
□RUN GFX2(1,"SELECT")
□RUN GFX2(PATH,"DWEND")
□CLOSE #PATH
□END
```

SETDPTR Set draw pointer

Syntax: **RUN GFX2([*path*,]"SETDPTR",*xcor,ycor*)**

Function: Places the draw pointer at the specified coordinates. The draw pointer selects the beginning point of the next graphics draw function (such as CIRCLE, LINE, BOX, and so on), if you do not supply other coordinates.

Parameters:

path	The route to the screen where you want to set the draw pointer.
xcor,ycor	The screen coordinates for the draw pointer location. The X-coordinates are in the range 0-639. The Y-coordinates are in the range 0-191.

Examples:

```
RUN GFX2("SETDPTR",100,5)
```

Sample Program:

This procedure uses coordinates from a DATA statement for setting the draw pointer to create a series of star shapes.

```
PROCEDURE star
□DIM X,Y,T,J:INTEGER
□PRINT CHR$(12)
□FOR J=1 TO 10
□READ X,Y
□RUN GFX2("SETDPTR",X+J,Y+J+J)
□FOR T=1 TO 5
□READ X,Y
□RUN GFX2("LINE",X+J,Y+J+J)
□NEXT T
□NEXT J
□DATA 320,46,440,146,200,84,440,84,200,146,320,46
□END
```

UNDLNON Underline characters on
UNDLNOFF Underline characters off

Syntax: **RUN GFX2([*path*,]"UNDLNON")**
RUN GFX2([*path*,]"UNDLNOFF")

Function: Enables or disables character underline. After you execute UNDLNON, all characters displayed are underlined until you execute UNDLNOFF. The default is UNDLNOFF.

Parameters:

path The route to the window where you want to use underline characters.

Examples:

```
RUN GFX2("UNDLNON")
RUN GFX2("UNDLNOFF")
```

BASIC09 Quick Reference

This chapter contains a quick reference of all BASIC09 commands, statements, and functions. It includes commands for programming, editing, and debugging, as well as the Commands mode commands.

The following chart lists all BASIC09 keywords that you can use in a procedure.

Statements and Functions

Command	Description
ABS	Returns the absolute value of a number.
ACS	Calculates the arccosine of a number.
ADDR	Returns an integer value which is the absolute memory address of a variable, array, or structure in a process's address space.
AND	Generates the logical AND of two Boolean values.
ASC	Returns the ASCII code of the first character in a string.
ASN	Calculates the arcsine of a number.
ATN	Calculates the arctangent of a number.
BASE	Sets the lowest array or data structure subscript in a procedure to either 0 or 1.
BYE	Ends execution of a procedure and terminates BASIC09.
CHAIN	Executes a module, passing arguments if appropriate.
CHD	Changes the current data directory.
CHR$	Returns the ASCII character represented by a specified integer.
CHX	Changes the current execution directory.
CLOSE	Deallocates the specified path to a file or device.
COS	Calculates the cosine of a number.

Command	Description
CREATE	Opens a path and establishes a new file on disk.
DATE$	Returns the computer's current date and time.
DEG	Causes BASIC09 to calculate angles in degrees.
DATA	Stores data in a procedure to be accessed by the READ statement.
DELETE	Deletes a file from disk.
DIM	Declares simple variables, arrays or complex data structure for size and type.
DO	See WHILE/DO/ENDWHILE.
ELSE	See IF/THEN/ELSE/ENDIF.
END	Terminates execution of a procedure. Returns to the calling procedure or to BASIC09's command mode. Displays the specified text.
ENDEXIT	See EXITIF/ENDEXIT.
ENDIF	See IF/THEN/ELSE/ENDIF.
ENDLOOP	See LOOP/ENDLOOP.
ENDWHILE	See WHILE/DO/ENDWHILE.
EOF	Tests for the end of a disk file.
ERR	Returns the error code of the most recent error.
ERROR	Generates the specified error.
EXITIF/ ENDEXIT	Tests conditions in a loop. The procedure exits the loop if the condition is true.
EXP	Calculates e (2.71828183) raised to the specified value.
FALSE	A Boolean function that always returns FALSE.
FIX	Rounds a real number and converts it to an integer.
FLOAT	Converts a byte or integer value to a real number.

Command	Description
FOR/NEXT	Creates a program loop of a specified number of repetitions.
GET	Reads an element or a data structure from a binary file or a device.
GOSUB/ RETURN	Transfers program control to a specified subroutine. RETURN sends execution back to the calling routine.
IF/THEN/ELSE/ ENDIF	Evaluates an expression and performs an operation if the conditions are met. Including ELSE causes an alternate operation if the conditions are false.
INKEY	Stores the character of a keypress in a string variable.
INPUT	Causes a procedure to accept input from the keyboard or other specified device.
INT	Returns the largest whole number less than or equal to the specified value.
KILL	*Unlinks* a procedure. (Removes it from BASIC09's directory.)
LAND	Performs a bit-by-bit logical AND on two-byte, or integer, values.
LEFT$	Returns the specified number of characters, from the leftmost portion of a string.
LEN	Returns the length of the specified string.
LET	Assigns a value to a variable.
LNOT	Performs a bit-by-bit logical NOT function on two-byte, or integer, values.
LOG	Calculates the natural logarithm.
LOG10	Calculates a base 10 logarithm.
LOOP/ ENDLOOP	Establishes a loop. Use EXITIF and ENDEXIT to test the loop and exit when a specified condition is true.
LOR	Performs a bit-by-bit logical OR on two-byte, or integer, values.

Command	Description
LXOR	Performs a bit-by-bit logical EXCLUSIVE OR on two-byte, or integer, values.
MID$	Returns the specified number of characters, beginning at the specified position in a string.
MOD	Returns the modulus (remainder) of a division operation.
NEXT	See FOR/NEXT.
NOT	Returns the logical complement of a Boolean value.
ON ERROR/ GOTO	Traps errors and transfers control to the specified line number.
ON/GOSUB	Evaluates an expression. Then, selects from a list the line number that is in the position indicated by the result of the expression. Procedure execution transfers to the selected line.
ON/GOTO	Evaluates an expression. Then, selects from a list the line number that is in the position indicated by the result of the expression. Procedure execute jumps to the selected line.
OPEN	Opens an I/O path to an existing file or device.
OR	Performs a logical OR on two Boolean values.
PARAM	Describes the parameters a called procedure expects from a calling procedure.
PAUSE	Suspends execution of a procedure, and enters the Debug mode.
PEEK	Returns the byte value of a memory address.
PI	Represents the constant 3.14159265.
POKE	Stores a byte value at a specified memory address.
POS	Returns the current character position of the print buffer.

Command	Description
PRINT	Sends the specified characters or values to the display.
PRINT USING	Sends characters or values to the display, using the specified format.
PRINT#	Sends the specified characters or values to the specified path.
PRINT# USING	Sends characters or values to the specified path using the specified format.
PUT	Writes data to a random access file.
RAD	Causes BASIC09 to calculate angles in radians.
READ	Accesses data from procedure DATA lines or from files or devices.
REM	Indicates that the following characters in a procedure line are comments and are not to be executed. Also use (* *), or (*.
REPEAT/UNTIL	Establishes a loop that executes until the specified condition is met.
RESTORE	Restores the DATA pointer to the first data item or to a specified line.
RETURN	See GOSUB/RETURN.
RIGHT$	Returns the number of characters specified, from the rightmost portion of a string.
RND	Returns a random number from a specified range.
RUN	Calls another procedure for execution.
SEEK	Changes the file pointer address.
SGN	Determines the sign of a number.
SHELL	Calls an OS-9 command or program for execution.
SIN	Calculates the sine of a specified value.
SIZE	Returns the number of bytes assigned to a variable, array, or complex data structure.
SQ	Calculates a value raised to the power of two.

Command	Description
SQR/SQRT	Calculates the square root of a positive number.
STEP	Sets the size of increment in a FOR/NEXT loop.
STOP	Terminates the execution of all procedures and returns to the BASIC09 Command mode.
STR$	Converts numeric data to string data.
SUBSTRING	Returns the starting position of a sequence of characters in a string.
SYSCALL	Executes an OS-9 System Call.
TAB	Begins a print operation at the specified column.
TAN	Calculates the tangent of a value.
TRIM$	Strips trailing spaces from the specified string.
TRON/TROFF	Turn the trace mode on and off.
TRUE	Returns the Boolean value of TRUE.
TYPE	Defines a new data type.
UNTIL	See REPEAT/UNTIL.
USING	See PRINT USING.
VAL	Converts a string to an integer.
WHILE/DO/ ENDWHILE	Executes a loop as long as a specified condition is true.
WRITE	Writes data in ASCII format to a file or device.
XOR	Performs a logical EXCLUSIVE OR on two Boolean values.

Commands by Type

Statements

BASE 0	DIM	GOSUB	OPEN	RETURN
BASE 1	ELSE	GOTO	PARAM	RUN
BYE	END	IF/THEN	PAUSE	SEEK
CHAIN	ENDEXIT	INPUT	POKE	SHELL
CHD	ENDIF	KILL	PRINT	STOP
CHX	ENDLOOP	LET	PUT	TROFF
CLOSE	ENDWHILE	LOOP	RAD	TRON
CREATE	ERROR	NEXT	READ	TYPE
DATA	EXITIF/THEN	ON ERROR/GOTO	REM	UNTIL
DEG	FOR/TO/STEP	ON/GOSUB	REPEAT	WHILE/DO
DELETE	GET	ON/GOTO	RESTORE	WRITE

Transcendental Functions

ACS	COS	LOG10	SIN
ASN	EXP	PI	TAN
ATN	LOG		

Numeric Functions

ABS	LAND	MOD	SQ
FIX	LNOT	RND	SQR
FLOAT	LOR	SGN	SQRT
INT	LXOR		

String Functions

ASC	LEFT$	RIGHT$	TRIM$
CHR$	LEN	STR$	VAL
DATE$	MID$	SUB	STR
INKEY			

Miscellaneous Functions

ADDR	FALSE	SIZE	SYSCALL
EOF	PEEK	TAB	
ERR	POS	TRUE	

Data Types

The following list shows the BASIC09 data type you can specify when defining a variable.

Type	Function
BOOLEAN	Returns TRUE or FALSE
BYTE	Specifies that a numeric variable is to store single-byte values.
INTEGER	Specifies that a numeric variable is to store integer (two-byte) values.
REAL	Specifies that a numeric variable is to store real (five-byte) values.
STRING	Specifies that a variable is to store ASCII characters.

Types of Access for Files

You can use the following parameters with the CREATE and OPEN commands. Check the individual commands for information on which parameter to use with which command.

Parameter	Function
DIR	Lets BASIC09 access a directory-type file for reading. Do not use with UPDATE or WRITE.
EXEC	Lets BASIC09 access the current execution directory rather than the current data directory.
READ	Sets the file access mode for reading.
WRITE	Sets the file access mode for writing.
UPDATE	Sets the file access mode for both reading and writing.

Command Mode

The following chart lists the commands available from the BASIC09 Commands mode:

Command	Function
$	Calls the shell command interpreter to execute an OS-9 command.
BYE or CTRL BREAK	Returns you to the OS-9 system or to the program that called BASIC09.
CHD	Changes the current data directory.
CHX	Changes the current execution directory.
DIR	Displays the name, size, and variable storage requirement of each procedure in the workspace.
EDIT or E	Enters the procedure editor/compiler mode.
KILL	Removes one or more procedures from the workspace.
LIST	Displays a formatted listing of one or more procedures.
LOAD	Loads all procedures from a file into the workspace.
MEM	Displays current workspace size or reserves a specified amount of memory for the workspace.
PACK	Performs a second compilation and stores the resulting file in the execution directory.
RENAME	Changes a procedure's name.
RUN	Causes a procedure to execute.
SAVE	Writes one or more procedures to disk.

Edit Commands

The following chart lists the commands available from the Edit mode:

Command	Function
ENTER	Moves the edit pointer to the next line.
+*num*	Moves the edit pointer forward a specified number of lines.
+*	Moves the edit pointer past the last line.
−*num*	Moves the edit pointer back a specified number of lines.
−*	Moves the edit pointer to the first line.
text	A space followed by text inserts an unnumbered line before the current line.
line	Typing a line number with or without text following it inserts the line into the procedure.
line ENTER	Moves the edit pointer to the line *line*.
c/*str1/str2*/	Changes the text *str1* to the text *str2*.
c*/*str1/str2*	Changes all occurrences of *str1* to *str2*.
d	Deletes the current line.
d*	Deletes all the lines in the procedure.
l	Lists the current line.
l*	Lists all the lines in the current procedure.
q	Terminates the edit session.
r	Renumbers lines from the first line number, in increments of 10.
r*	Renumbers all numbered lines in increments of 10. The first line number is 100.
r *line*	Renumbers lines from *line* in increments of 10.
r *line num*	Renumbers lines from *line*, in increments of *num*.
s/*str*	Searches for the first occurrences of *str*.
s*/*str*	Searches for all occurrences of *str*.

Debug Commands

The following table lists all the Debug commands and what they accomplish:

Command	Function
$*command*	Tells BASIC09 to execute the specified OS-9 command or program.
BREAK	Sets a breakpoint at the specified procedure.
CONT	Causes procedure execution to continue.
DEG/RAD	Selects either degrees or radians as the unit of angle measurement for trigonometric functions.
DIR	Displays the procedures in the workspace.
Q	Leaves the Debug mode for the System mode.
LET	Assigns a new value to a variable.
LIST	Displays a source listing of the suspended procedure.
PRINT*var*	Displays the value of the specified variable.
STATE	Lists the *nesting* order of all active procedures.
STEP*num*	Causes execution of the suspended procedure in specified increments.
TRON/TROFF	Turns the trace function on and off.

BASIC09 Command Reference

BASIC09 is made of keywords (functions and statements) that you use, with their parameters, to instruct the computer to perform certain operations.

This chapter is a complete reference for all of BASIC09's keywords.

Keyword Format

The reference to each keyword is organized in this manner:

- The keyword.
- The proper *syntax* (spelling and form) for using the keyword.
- A brief description of the keyword's purpose or effect.
- Descriptions of any parameters or arguments for the keyword.
- Notes about special features or requirements of the keyword, when appropriate.
- One or more examples for using the keyword.
- One or more sample procedures.

This format can vary slightly, depending on the complexity of each keyword. For instance, some keywords require parameters or arguments, and others do not. Some keywords are self-explanatory and do not require a sample procedure.

The Syntax Line

The second line in each command or keyword reference is the syntax line. This line uses keyword *constants* and keyword *variables* to show you how to construct a command line. Constants are words, numbers, or symbols that you type exactly as they appear. Variables are words that only represent the actual words, numbers, or symbols that you must supply for the command.

All variables are italic. When you see an italicized word, you know that you must supply some other word, name, symbol, or value in place of that word. If a word, symbol, or value is not italicized, type it exactly the way it appears in the syntax line.

The syntax line also uses symbols to help you understand how to construct a command line. These symbols are:

[ ] Words, names, value, or symbols contained between right and left brackets are optional. You can use them or not, depending on what you want to accomplish with the command.

... Ellipsis indicates that the last parameter can be repeated.

The following syntax line for DELETE requires only one parameter, the variable *pathname*.

```
DELETE "pathname"
```

Because *pathname* is italicized, you know that you must replace it with other text—in this case the pathlist to the file you want to delete. If you wanted to delete a file named Test from the ROOT directory of Drive /D1, this syntax line tells you that you must type:

```
delete "/d1/test"
```

Other syntax lines are more complex, such as the line for CREATE:

```
CREATE #path,"pathlist" [access mode]
[+access mode][+...]
```

This line tells you how to create a path to a file or device. Because the number symbol (#) is not italicized, you type it after the blank space following the keyword. However, *path,pathlist*, and *access mode* are all italicized. You must replace them with other names or values.

The *access mode* variable is contained within brackets. This tells you that it is optional. You can include an access mode, or not. If you don't, BASIC09 opens the path in the Update Mode.

The second *access mode* shows that the command allows two access mode parameters, preceded by a plus symbol. The ellipsis show that you can have even more *access mode* parameters.

Other syntax lines show that no parameters are required, such as:

```
DATE$
```

This command returns the current date. There is nothing it requires, and you can do nothing else with it.

Sample Programs

The sample programs in this chapter are complete. That is, you can type them, run them, and get a result. The procedures let you see the syntax and form of a command, as well as showing you how it might be used in a program.

Because the programs are executable, the manual shows unformatted listings (without relative address, indented control structures, and so on). This helps eliminate confusion for you when you type the program. You can type it exactly as it appears, exit the editor, and run the procedure.

ABS Return absolute value

Syntax: ABS(*number*)

Function: Computes the absolute value of *number*. A number's absolute value is its magnitude without regard to its sign. Absolute values are always positive or zero.

Parameters:

number Any positive or negative number.

Examples:

```
PRINT ABS(-66)

X=ABS(Y)
```

Sample Program:

The following procedure asks you to type the temperature, and makes an appropriate comment. It uses ABS to get the absolute value of the temperature.

```
PROCEDURE temperature
□DIM TEMP:INTEGER
□INPUT "What's the temperature outside? (Degrees
F)...",TEMP
□IF TEMP<0 THEN
□PRINT "That's "; ABS(TEMP); " below
zero!□□□Brrrrrrr!"
□END
□ENDIF
□IF TEMP=0 THEN
□PRINT "Zero degrees? That's mighty cold!"
□END
□ENDIF
□PRINT TEMP; " degrees above zero? That's kind of
balmy..."
□END
```

ACS Return arccosine

Syntax: **ACS(*number*)**

Function: Calculates the arccosine of *number*. Use the DEG or RAD commands to tell BASIC09 if *number* is in degrees or radians. If you do not specify degrees or radians, the default is radians.

Parameters:

number The number for which you want to compute the arccosine.

Examples:

```
PRINT ASC(.6561)
```

Sample Program:

The procedure calculates the arccosine of a value you type and expresses the result in degrees.

```
PROCEDURE arccosine
□DEG
□DIM NUM:REAL
□INPUT "Enter a number between -1 and 1",NUM
□PRINT "The arccosine of "; NUM; " is---";
ACS(NUM)
□END
```

ADDR Return the location of a variable

Syntax: **ADDR(*name*)**

Function: Returns the absolute location in a process's address space of the variable, array, or data structure assigned to *name*. The address returned is that of the first character in the variable. If the variable is numeric, one or more of the locations might contain zero.

For instance, if you use ADDR to obtain the address of an integer variable that contains the value 44, the first address location (byte) contains 0, and the second location contains 44.

Parameters:

name	The name of a string, a numeric variable, an array, or a data structure.

Examples:

This procedure displays the memory address where a variable named X resides:

```
PRINT ADDR(X)
```

Sample Program:

This procedure uses ADDR to tell you the memory location of the variable that stores your keyboard entry.

```
PROCEDURE address
□DIM A:INTEGER
□DIM TEST:STRING
□INPUT "Type a string of characters...",TEST
□A=ADDR(TEST)
□PRINT "The string you typed is stored at address
"; A
□PRINT "This is what it contains:..."
□FOR T=A TO A+LEN(TEST)
□PRINT CHR$(PEEK(T));
□NEXT T
□PRINT
□END
```

AND Performs a logical AND operation

Syntax: *operand1* **AND** *operand2*

Function: Performs the logical AND operation on two or more values, returning a value of either TRUE or FALSE.

Parameters:

operand1 *operand2*	Can be either numeric or string values.

Examples:

```
PRINT A>3 AND B>3

PRINT A$="YES" AND B$="YES"
```

Sample Program:

The following program calculates an insurance premium rate that is based on the answers to some lifestyle questions. Every time you press [Y], the premium rate goes up. The procedure uses AND to increase the rate by two percent if you both smoke and drink.

```
PROCEDURE policy
□DIM POLICY_VALUE,RATE:REAL
□DIM SMOKE,DRINK:STRING[1]
□POLICY_VALUE=1000000.
□RATE=.001
□INPUT "Do you smoke? (Y/N)...",SMOKE
□INPUT "Do you drink? (Y/N)...",DRINK
□IF SMOKE="Y" AND DRINK="Y" THEN RATE=RATE+.02
□ELSE
□IF SMOKE="Y" THEN RATE=RATE+.01
□ENDIF
□IF DRINK="Y" THEN RATE=RATE+.01
□ENDIF
□ENDIF
□PRINT "Your premium is "; RATE*POLICY_VALUE
□END
```

ASC **Returns ASCII code**

Syntax: ASC(***string***)

Function: Returns the ASCII code for the first character of *string*.

ASC returns the value as a decimal number. If *string* is null (contains no characters) BASIC09 returns Error 67 (Illegal Argument).

Parameters:

string Any string type variable or constant.

Examples:

```
PRINT ASC("Hello")

X = ASC(A$)
```

Sample Program:

The following procedure determines whether the first character you enter is a hexadecimal digit. To do this, it gets the ASCII value of the character and compares it to the ranges for characters between 1 and 0 and A and F.

```
PROCEDURE hexcheck
□DIM A:INTEGER
□DIM HEXNUM:STRING
□LOOP
□INPUT "Enter a hexadecimal value...",HEXNUM
□A=ASC(HEXNUM) \            (* GET THE ASCII CODE *)
□EXITIF A<48 OR A>57 AND A<65 OR A>70 THEN
□PRINT "Not a hex number."
□END
□ENDEXIT
□PRINT "Ok."
□ENDLOOP
□END
```

ASN Returns arcsine

Syntax: ASN(***number***)

Function: Calculates the arcsine of *number*. ASN expresses its result in radians unless you specify otherwise (see DEG).

Parameters:

number The number for which you want to calculate the arcsine.

Examples:

```
PRINT ASC(.6561)
```

Sample Program:

The following program calculates the arcsine of a number you enter and expresses the result in degrees.

```
PROCEDURE arcsine
□DIM NUM:REAL
□DEG
□INPUT "Enter a number (-1 to 1) ",NUM
□PRINT "The arcsine of a "; NUM; " is---";
ASN(NUM)
□END
```

ATN **Returns arctangent**

Syntax: ATN(***number***)

Function: Calculates the arctangent of *number*.

Parameters:

number	The number for which you want to find the arctangent.

Examples:

```
PRINT ASC(.6561)
```

Sample Program:

This procedure calculates arcsine, arccosine, and arctangent for a value you enter.

```
PROCEDURE anglecalc
□DIM NUM:REAL
□DEG
□INPUT "Enter a number ",NUM
□PRINT
□PRINT " ","Arcsine","Arccosine","Arctangent"
□PRINT "Number","Degrees","Degrees","Degrees"
□PRINT "-----------------------------------------
--------------------"
□IF NUM>1 OR NUM<-1 THEN
□PRINT NUM,"UNDEF","UNDEF",ATN(NUM)
□PRINT
□END
□ENDIF
□PRINT NUM,ASN(NUM),ACS(NUM),ATN(NUM)
□PRINT
□END
```

BASE Set array base

Syntax: **BASE 0**
BASE 1

Function: Sets a procedure's lowest array or data structure index to either 0 or 1. If you want to have the first elements in arrays set to 0, you must include `BASE 0` at the beginning of the procedure.

The BASE statement does not affect string operations such as MID$, RIGHT$, and LEFT$. BASIC09 always indexes the first character of a string as 1.

Parameters:

0 or 1	If you do not indicate a BASE setting in a procedure, BASIC09 uses a default of 1.

Examples:

```
BASE 0
```

Sample Program:

This procedure determines how many times RND selects each number between 0 and 11 out of 1000 selections. It stores the results in an array of 12 elements. Because it specifies BASE 0, one of the elements in the array is 0. Whenever the procedure picks a random number, it increments the value in the corresponding array number by one.

```
PROCEDURE randomtest
□BASE 0                             (* set the array base at 0.
□DIM RND_ARRAY(12),X,R:INTEGER (* dimension array to hold results.

□FOR X=0 TO 11
□RND_ARRAY(X)=0                     (* initialize array elements at zero.
□NEXT X
□SHELL "TMODE -PAUSE"               (* turn off screen pause.

□FOR X=1 TO 1000
□R=RND(11)                          (* select random number 1000 times.
```

```
□RND_ARRAY(R)=RND_ARRAY(R)+1
                                  (* add 1 to appropriate element.

□PRINT 1001-X                      (* count down from 1000 to 1.

□NEXT X

□FOR X=0 TO 11
□PRINT "RND selected "; X; " "; RND_ARRAY(X); "
times."                           (*display array

□NEXT X
□SHELL "TMODE PAUSE"               (* turn scroll lock back on.
□END
```

BYE End procedure, terminate BASIC09

Syntax: **BYE**

Function: Ends execution of a procedure and terminates BASIC09. The statement closes any open files, but you lose any unsaved procedures or data.

Use BYE to exit packed programs that you call from OS-9 and especially programs that you call from procedure files.

Parameters: None

Examples:

```
INPUT "Press ENTER to return to the system.";Z$
BYE
```

Sample Program:

This procedure calculates the payments and interest of a loan. When it is through, it exits the procedure and BASIC09 with a BYE statement.

```
PROCEDURE loan
□DIM PRIN,LENG,RATE,MONPAY:REAL
□DIM RESPONSE:STRING[1]
□REPEAT
□PRINT "Amortization Program"
□INPUT "How much do you want to borrow?...",PRIN
□INPUT "For how many months?...",LENG
□INPUT "At what interest rate?...",RATE
□A=RATE/1200
□B=1-1/(1+A)^LENG
□MONPAY=PRIN*A/B
□MONPAY=INT(MONPAY*100+.5)/100
□PRINT "Monthly payments are...$";
□PRINT USING "R12.2<",MONPAY
□PRINT "The total interest to pay is...$";
□PRINT USING "r12.2<",MONPAY*LENG-PRIN
□PRINT
□INPUT "Do another calculation?...",RESPONSE
□PRINT
□PRINT
□UNTIL RESPONSE<>"Y"
□BYE
□END
```

CHAIN **Execute another module**

Syntax: **CHAIN "*module* [*parameters*][...]"**

Function: CHAIN performs an OS-9 chain operation, passing *module* as the name of a program to execute. If you include other parameters, CHAIN passes them to the executing module. The module must be programmed to expect parameters of the type you provide.

CHAIN exits BASIC09, unlinks BASIC09, and returns the freed memory to OS-9.

CHAIN can begin execution of any module, not only BASIC09 modules. It executes the module indirectly through the shell in order to take advantage of the shell's parameter processing. This has the side effect of leaving the initiated shells active. Programs that repeatedly chain to each other eventually fill memory with waiting shells. To prevent this, use the EX option to initialize a shell.

BASIC09 does not close files that are open when you execute CHAIN. However, the OS-9 FORK call passes only the standard I/O paths (0, 1, and 2) to a child process. Therefore, if you need to pass an open path to another program segment, use the EX shell option.

Parameters:

module	The name of the procedure module you want BASIC09 to execute.
parameters	String data passed to the *chained* module.

Examples:

```
CHAIN "ex BASIC09 menu"

CHAIN "BASIC09 #10k sort (""datafile"",
""tempfile"")"

CHAIN "DIR /D0"

CHAIN "Dir; Echo *** Copying Directory ***; ex
basic09 copydir"
```

Sample Program:

This procedure chains to two others to display a directory or a file. It uses CHAIN to call the procedures.

```
PROCEDURE chaining
□DIM RESPONSE:BYTE
□PRINT USING "S26^","- MENU -" (* print menu title.
□PRINT
□PRINT "1. List current data directory" (* print menu.
□PRINT "2. Display a file"
□PRINT "3. Exit to system"
□PRINT
□INPUT "Select a function (1-3) ",RESPONSE (* function you want.
□ON RESPONSE GOTO 100,200,300 (* select appropriate function.
100□CHAIN "EX BASIC09 dirlook" (* chain to list directory.
200□CHAIN "EX BASIC09 display" (* chain to list file.
300□BYE

PROCEDURE dirlook
□REM Lists the specified directory

□SHELL "DIR"          (* execute dir command.
□CHAIN "EX BASIC09 chaining" (* chain back to calling proc.
□END

PROCEDURE display
□REM Lists the specified file.

□DIM FILE,JOB:STRING
□INPUT "Path of file to display...",FILE
□JOB="LIST "+FILE
□SHELL JOB          (* list specified file.
□CHAIN "EX BASIC09 chaining" (* chain back to calling proc.
□END
```

CHD Change data directory
CHX Change execution directory

Syntax: **CHD *dirpath***
CHX *dirpath*

Function: Changes the current data or execution directory.

Parameters:

dirpath An existing data or execution directory.

Examples:

```
CHD "/D1/ACCOUNTS/RECEIVABLE"

CHX "/D1/CMDS"

CHD ".."
```

Sample Program:

This procedure creates a directory, and makes it the data directory. Then, it creates a file in the new directory, exits the new directory, and deletes the file and the directory.

```
PROCEDURE chdtest
□DIM PATH:BYTE
□SHELL "MAKDIR TEST"             (* create new directory named TEST.
□CHD "TEST"                      (* make TEST the data directory.

□CREATE #PATH,"samplefile":WRITE (* create a file in TEST.
□REM      Write data into the new file
□WRITE #PATH,"This file is for testing only."
□WRITE #PATH,"It will be destroyed when this procedure ends."
□CLOSE #PATH

□SHELL "LIST samplefile"         (* list the new file.
□CHD ".."                        (* make the ROOT the data directory.
□SHELL "DEL TEST/samplefile"     (* delete the file.
□SHELL "DELDIR TEST"             (* delete the directory.
□END
```

CHR$ Return ASCII character

Syntax: **CHR$(*code*)**

Function: Returns the ASCII character for the value of *code*. CHR$ is the inverse of the ASC function, which returns the ASCII code for a given character. For a complete listing of ASCII codes, see Chapter 9.

Parameters:

code The ASCII value for a keyboard character or special block graphics character.

Examples:

```
PRINT CHR$(88)
```

Sample Program:

By increasing by one the ASCII values of characters you type, the following program creates a secret code. It uses CHR$ to display the secret code.

```
PROCEDURE secret
□DIM TEXT,SECRETLINE:STRING[80]
□DIM T,CODECHAR:INTEGER
□TEXT=""
□SECRETLINE=""

□PRINT "Type a line to code in capital letters..."
□INPUT TEXT                    (* you type a line.
□FOR T=1 TO LEN(TEXT)
□CODECHAR=ASC(MID$(TEXT,T,1)) (* look at each character in line.
□IF CODECHAR=90 THEN           (* is it "Z"? If yes then
□CODECHAR=64                   (* make it one less than "A".
□ENDIF
□IF CODECHAR=32 THEN           (* is character a space? If yes then
□CODECHAR=31                   (* decrease its value by one.
□ENDIF

□SECRETLINE=SECRETLINE+CHR$(CODECHAR+1) (* add 1 to characters.
□NEXT T
□PRINT SECRETLINE              (* print the secret code.
□END
```

CHX Change execution directory
CHD Change data directory

Syntax: **CHX *dirpath***
CHD *dirpath*

Function: Changes the current execution or data directory.

Parameters:

dirpath An existing execution or data directory.

Examples:

```
CHX "/D1/CMDS"

CHD "/D1/ACCOUNTS/RECEIVABLE"

CHD ".."
```

CLOSE Deallocate file or device path

Syntax: **CLOSE #*pathnum***

Function: Deallocates the file or device path specified by *pathnum*.

When you OPEN or CREATE a file, BASIC09 allocates a path number to the variable you supply in the OPEN or CREATE command. The system then *knows* the path by that number. If the path you CLOSE is to a non-shareable device (such as a printer), the system releases the device for other use. Do not close paths 0, 1, and 2 (the standard I/O paths) unless you immediately open a new path to take over the standard path number.

Parameters:

pathnum The name of variable containing the path number or the actual number of the path to a file or device.

Examples:

```
CLOSE #FILEPATH, #PRINTERPATH, #TERMPATH

CLOSE #5, #6, #7

CLOSE #1       \ (* closes the standard output path *)
OPEN #PATH,"/T1"    \ (* redirects standard output *)
```

Sample Program:

This procedure creates a directory named TEST and changes it to the data directory. It then creates a file named Samplefile and writes data to the file. Finally it changes back to the parent directory and deletes Samplefile and TEST.

```
PROCEDURE close
□DIM PATH:BYTE
□SHELL "MAKDIR TEST"
□CHD "TEST"
□CREATE #PATH,"samplefile":WRITE (* create a new file.
□WRITE #PATH,"This file is for testing only."
□WRITE #PATH,"It will be destroyed when this procedure ends."
□CLOSE #PATH (* close the file.
□SHELL "LIST samplefile"
□CHD ".."
□SHELL "DELDIR TEST"
□END
```

COS Return cosine

Syntax: COS(***number***)

Function: Calculates the cosine of *number*. Unless you specify DEG, COS interprets the value of *number* in radians.

Parameters:

number The number for which you want to find the cosine.

Examples:

```
PRINT COS(45)
```

Sample Program:

This procedure calculates sine, cosine, and tangent of a value you enter.

```
PROCEDURE ratiocalc
□DIM NUM:REAL
□DEG
□INPUT "Enter a number...",NUM
□PRINT
□PRINT "Number","SINE","COSINE","TAN"
□PRINT "---------------------------------------
--------------------"
□PRINT ANGLE,SIN(NUM),COS(NUM),TAN(NUM)
□PRINT
□END
```

CREATE **Establish a disk file.**

Syntax:
CREATE *#path,"pathlist"* [*access mode*]
[+*access mode*][+...]

Function: Creates a file on a disk. When you create a file, you can select one or more of the following access modes for the file:

Mode	Function
READ	Lets you read (receive) data from a file but does not let you write (send) data to the file.
WRITE	Lets you write data to a file but does not let you read data from a file.
UPDATE	Lets you both read from and write to a file.

Parameters:

path	The name of the variable in which BASIC09 stores the number of the opened path.
pathlist	The route to the file or device to be opened, including the filename, if appropriate.
access mode	The type of access to be allowed for the file or device. Use plus symbols to allow more than one type of access with a single file.

Notes:

- You can access files either sequentially or randomly. With random access, you must establish the filing system you want for a particular application.
- Files are byte-addressed, and you are not restricted by explicit record lengths. You can read the data one byte at a time, or in whatever size portions you want.

- A new file has a size of zero. OS-9 then expands the file automatically when PRINT, WRITE, or PUT statements write beyond the current end-of-file.

Examples:

```
CREATE #TRANS,"transportation":UPDATE

CREATE #SPOOL,"/user4/report":WRITE

CREATE #OUTPATH,name$:UPDATE+EXEC
```

Sample Program:

This procedure CREATEs a directory named TEST and makes it the data directory. It creates a file in TEST named Samplefile, writes data to the file, then resets the parent directory as the data directory. Finally, it deletes Samplefile and TEST.

```
PROCEDURE close
□DIM PATH:BYTE
□SHELL "MAKDIR TEST"
□CHD "TEST"
□CREATE #PATH,"samplefile":WRITE (* create a file.
□WRITE #PATH,"This file is for testing purposes only."
□WRITE #PATH,"It will be destroyed when this procedure ends."
□CLOSE #PATH (* close the file.
□SHELL "LIST samplefile"
□CHD ".."
□SHELL "DELDIR TEST"
□END
```

DATA Store numeric and string information

Syntax: DATA "*item*"[,"*item*",...]

Function: Stores numeric and string constants to be accessed by a READ statement. A DATA line can contain up to 254 characters. Each item in the list must be separated by commas.

You can place DATA statements anywhere in a procedure that is convenient. BASIC09 reads sequentially, starting with the first item in the first DATA statement, and ending with the last item in the last DATA statement.

The following rules apply to data items:

- You must place all string data between quotation marks.
- To include quotes in string-type data, use consecutive quotation marks, like this: `DATA "He said, ""go home"" to me"`.
- You can use RESTORE to reset the data *pointer*. Using RESTORE without an argument resets the pointer to the beginning of the data items. Using RESTORE with a line number, resets the pointer to the first item in the specified line.
- The READ statement can support a list of one or more variable names of various types. The data types in DATA statements must match the variable types used in the corresponding READ statements.
- You can include arithmetic expressions in data items. READ causes the expressions to be evaluated and returns the result of the expression as the data item.

Parameters:

item	Numeric or string characters. Enclose string characters in quotation marks.

Examples:

```
DATA 1.1,1.5,9999,"CAT","DOG"

DATA SIN(TEMP/25), COS(TEMP*PI)

DATA TRUE,FALSE,TRUE,TRUE,FALSE

DATA "The rain in spain","falls mainly on the
plain"
```

Sample Program:

This procedure calculates the day of the week for a date you enter. A data statement contains the names of the weekdays.

```
PROCEDURE weekday
□DIM X,DAY,MONTH,YEAR,CALC:INTEGER
□DIM ANUM,BNUM,CNUM,DNUM,ENUM,FNUM,GNUM,HNUM,INUM:
INTEGER
□DIM WEEKDAY(7):STRING[9]
□PRINT USING "S60^","Day of the Week Program"
□PRINT USING "S60^","For any year after 1752"
□PRINT
□INPUT "Enter day of the month as two digits, such
as 08...",DAY
□INPUT "Enter month as two digits, such as
12...",MONTH
□INPUT "Enter year as four digits, such as
1986...",YEAR
□FOR X=1 TO 7
□READ WEEKDAY(X)
□NEXT X
□ANUM=INT(.6+1/MONTH)
□BNUM=YEAR-ANUM
□CNUM=MONTH+12*ANUM
□DNUM=BNUM/100
□ENUM=INT(DNUM/4)
□FNUM=INT(DNUM)
□GNUM=INT(5*BNUM/4)
□HNUM=INT(13*(CNUM+1)/5)
□INUM=HNUM+GNUM-FNUM+ENUM+DAY-1
□INUM=INUM-7*INT(INUM/7)+1
□PRINT
□PRINT "The day of the week on "; DAY; "/"; MONTH;
□PRINT "/"; YEAR; " is..."; WEEKDAY(INUM)
□DATA "Sunday","Monday","Tuesday","Wednesday",
"Thursday"
□DATA "Friday","Saturday"
□END
```

DATE$ Provide date and time

Syntax: DATE$

Function: Returns the date and time. The OS-9 internal date is kept in the format:

year/month/day hour:minutes:seconds

If your OS-9 Startup file contains the SETIME command, the system asks you to enter the date and time whenever it boots. If it does not contain the SETIME command, the date and time start from 86/09/01:00:00:00.

You can use the normal string functions to access the data contained in DATE$, but you cannot use functions or operations that attempt to change or append to its values. To reset the date or time or both, use the SHELL command, such as:

```
SHELL "SETIME"
```

Parameters: None

Examples:

```
PRINT DATE$
```

Sample Program:

This program is essentially the same as the sample program for the DATA statement, except that it gets the day, month, and year from DATE$.

```
PROCEDURE date
□DIM X,DAY,MONTH,YEAR,CALC:INTEGER
□DIM ANUM,BNUM,CNUM,DNUM,ENUM,FNUM,GNUM,HNUM,INUM:INTEGER
□DIM WEEKDAY(7):STRING[9]
□MONTH=VAL(MID$(DATE$,4,2)) (* get month from DATE$.
□DAY=VAL(MID$(DATE$,7,2)) (* get day from DATE$.
□YEAR=VAL("19"+LEFT$(DATE$,2)) (* get year from DATE$.

□FOR X=1 TO 7
□READ WEEKDAY(X)
```

```
□NEXT X
□ANUM=INT(.6+1/MONTH)
□BNUM=YEAR-ANUM
□CNUM=MONTH+12*ANUM
□DNUM=BNUM/100
□ENUM=INT(DNUM/4)
□FNUM=INT(DNUM)
□GNUM=INT(5*BNUM/4)
□HNUM=INT(13*(CNUM+1)/5)
□INUM=HNUM+GNUM-FNUM+ENUM+DAY-1
□INUM=INUM-7*INT(INUM/7)+1
□PRINT
□PRINT "Today is "; WEEKDAY(INUM)

□DATA "Sunday","Monday","Tuesday","Wednesday","Thursday","Friday"
□DATA "Saturday"
□END
```

DEG Return trigonometric calculations in degrees

Syntax: **DEG**

Function: Causes a procedure to calculate trigonometric values in degrees. If you do not include the DEG statement, procedures produce radian values.

Parameters: None

Examples:

```
DEG
```

Sample Program:

This procedure calculates the sine, cosine, and tangent for a value you enter. Because it uses the DEG statement, it displays the results in degrees.

```
PROCEDURE degcalc
□DIM NUM:REAL
□DEG
□INPUT "Enter a number...",NUM
□PRINT
□PRINT "Number","SINE","COSINE","TAN"
□PRINT "----------------------------------------
-------------------"
□PRINT NUM,SIN(NUM),COS(NUM),TAN(NUM)
□PRINT
□END
```

DELETE Erase a disk file

Syntax: **DELETE "*pathname*"**

Function: DELETE removes a file from disk storage and releases the portion of the disk on which it resides. When you DELETE a file, it is permanently lost.

Parameters:

pathname — The complete pathlist to the file you want to delete, including the drive and one or more directories, if appropriate. You must surround the pathlist with quotation marks.

Examples:

```
DELETE "myfile"

DELETE "/D1/ACCOUNTS/receivables"
```

Sample Program:

This procedure creates a file named Samplefile, writes data to the file, then closes it. It then lists the file before deleting it.

```
PROCEDURE close
□DIM PATH:BYTE
□CREATE #PATH,"samplefile":WRITE (* create a file.
□WRITE #PATH,"This file is for testing purposes only."
□WRITE #PATH,"It will be destroyed when this procedure ends."
□CLOSE #PATH (* close the file.
□SHELL "LIST samplefile"
□DELETE "samplefile"
□END
```

DIM Assign variable storage

Syntax: **DIM *variable*[,...][:*type*][;*variable*][,...][:*type*][...]**

Function: Assigns storage space and declares types for variables, arrays, or complex data structures.

Parameters:

variable	A simple variable, an array structure, or a complex data structure.
type	BYTE, INTEGER, REAL, BOOLEAN, STRING, or user defined.

Notes:

- You declare simple arrays with DIM by using the variable name, without a subscript. If you do not explicitly declare variables, the system makes them type real unless they are followed by a dollar sign ($). The system dimensions variables ending with a dollar sign ($) as strings, with a length of 32 bytes. You must declare types of all other simple variables as to type.
- You can declare several variables of the same type by separating them with commas. To separate variables of different types, follow each type group with a colon, the type name, and then a semicolon.
- Define a maximum length for a string variable by enclosing the length in brackets following the type, like this:

```
DIM name:string[25]
```

If you do not define a maximum length, BASIC09 uses a default length of 32 characters. You can declare a shorter length or a longer length, up to the capacity of BASIC09's memory. If you try to extend a string beyond its declared length, or beyond the default length, the system ignores all extra characters. Thus the following:

```
DIM name:string[10]
name = "Abbernathinsky"
```

produces the string:

```
Abbernathi
```

- Arrays can have one, two, or three dimensions. The DIM format for dimensioned arrays is the same as for simple variables, except that you must follow each array name with a subscript, enclosed in parentheses, to indicate its size. The maximum array size is 32767.

  Arrays can be either of the standard BASIC09 type or of a user-defined type. For information on creating your own types for simple variables, arrays, and complex data structures, see TYPE.

Examples:

```
DIM logical:BOOLEAN

DIM a,b,c:INTEGER

DIM name,address,zip:STRING

DIM name:STRING[25]; address:STRING[30];
zip:INTEGER

DIM no1,no2,no3:REAL;no4,no5,no6:INTEGER;
no7:BYTE
```

Sample Program:

This procedure randomly selects letters and vowels to create six-letter words that might look like alien names. It first DIMs nine string variables to contain the letters selected for each name. It DIMs two integer variables to provide a loop counter and to store the number of names you request.

When asked, type the number of names you want to have the procedure generate.

```
PROCEDURE alien
□DIM B,BEGIN,F,FINISH:STRING
□DIM VOWELS,VOWEL1,VOWEL2:STRING
□DIM MID1,MID2:STRING
□DIM T,RESPONSE:INTEGER
□VOWELS="aeiouy"

□INPUT "How many alien names do you want to
see?...",RESPONSE
□BEGIN="ABCDFGHJKLMNPRSTVWXZ"
□FINISH="ehlmnprstvwyz"

□FOR T=1 TO RESPONSE
□B=MID$(BEGIN,RND(19)+1,1)
□F=MID$(FINISH,RND(12)+1,1)
□MID1=CHR$(RND(25)+97)
□MID2=CHR$(RND(25)+97)
□VOWEL1=MID$(VOWELS,RND(5)+1,1)
□VOWEL2=MID$(VOWELS,RND(5)+1,1)
□PRINT B; VOWEL1; MID1; MID2; VOWEL2; F,
□NEXT T

□PRINT
□END
```

DO Execute procedure lines in a loop

Syntax: **WHILE** ***expression*** **DO**
proclines
ENDWHILE

Function: Establishes a loop that executes the procedure lines between DO and ENDWHILE as long as the result of the expression following WHILE is true. Because the loop is tested at the top, the lines within the loop are never executed unless *expression* is true.

Parameters:

expression	A Boolean expression (produces a result of True or False).
proclines	Are program lines to execute if the expression is true.

See WHILE/DO/ENDWHILE for more information.

ELSE **Execute alternate action**

Syntax: **IF *condition* THEN**
action
ELSE
secondary action
ENDIF

Function: ELSE provides access to a secondary action within an IF/THEN test. When the condition tested by IF is **not** true, BASIC09 executes the *secondary action* preceded by ELSE.

Parameters:

condition	A Boolean expression (produces a result of True or False).
action	A line number to which the procedure is to transfer execution, or a program statement. If *action* is a line number, do not include the ENDIF statement in the IF test.
secondary action	One or more program statements.

For more information, see IF/THEN/ELSE

END Terminate a procedure

Syntax: **END [“*text*”]**

Function: Ends procedure execution and returns to the calling procedure, or to the highest level procedure. If you provide output text for END, it functions in the same manner as PRINT. You can use END several times in the same procedure. END is not required as the last statement in a procedure.

Parameters:

text A literal string or a string-type variable.

Examples:

```
END "Program Terminated"

LAST$="Session over"
END LAST$
```

Sample Program:

This procedure calculates a loan's term, using END to terminate routines.

```
PROCEDURE loaner
□DIM YOUPAY,PRINCIPLE,INTEREST,NUMPAY,YEARS,
MONTHS:REAL
□DIM RESPONSE:STRING[1]
□REPEAT
□PRINT
□PRINT USING "S45^","Loan Terms"
□PRINT
□INPUT "    Amount of Regular Payments...",YOUPAY
□INPUT "    Enter the Principle...",PRINCIPLE
□INPUT "    Enter the Annual Interest Rate...",
 INTEREST
□INPUT "    Enter the Number of Payments
 Yearly...",NUMPAY
```

```
□YEARS=-(LOG(1-PRINCIPLE*(INTEREST/100)/
(NUMPAY*YOUPAY))/(LOG(1+INTEREST/100/NUMPAY)*
NUMPAY))
□MONTH=INT(YEARS*12+.5)
□YEARS=INT(MONTH/12)
□MONTH=MONTH-YEARS*12
□PRINT "   The Term of Your Loan is "; YEARS; "
years and "; MONTH; " months."
□INPUT "Calculate another or Quit (C/Q)?...",
RESPONSE
□UNTIL RESPONSE<>"C" AND RESPONSE<>"c"
□END "Goodbye...I hope I helped you."
```

ENDEXIT Leave loop if a condition is True

Syntax: **EXITIF** ***condition*** **THEN**
proclines
ENDEXIT

Function: ENDEXIT terminates an EXITIF test. You always use EXITIF/THEN/ENDEXIT inside a procedure loop. If the Boolean expression tested by EXITIF is true, BASIC09 executes the program statements between THEN and ENDEXIT and then transfers program operation outside the loop. If the condition tested by EXITIF is not true, loop execution continues at the statement following ENDEXIT.

Parameters:

condition	A comparison operation that returns either True or False, such as A=B, A<B, or A=B=C.
proclines	One or more statements to perform if the Boolean expression tested by EXITIF is True.

For more information, see EXITIF/THEN/ENDEXIT

ENDIF Close IF statement

Syntax: **IF *condition* THEN**
action
[ELSE
***secondary action*]**
ENDIF

Function: ENDIF terminates an IF/THEN condition test. If the condition tested by IF is true, BASIC09 executes the statements between THEN and ENDIF. If the condition tested by IF is *not* true, BASIC09 transfers execution to the procedure line following ENDIF or (optionally) executes the statements following ELSE.

Parameters:

condition	A Boolean expression (produces a result of True or False).
action	A line number to which the procedure is to transfer execution. *Action* can also be a program statement. If *action* is a line number, do not include the ENDIF statement in the IF test.
secondary action	A program statement.

For more information, see IF/THEN/ELSE/ENDIF.

ENDLOOP Close LOOP statement

Syntax: **LOOP**
statement(s)
ENDLOOP

Function: ENDLOOP terminates a procedure loop established by the LOOP command. BASIC09 endlessly executes all procedure statements between LOOP and ENDLOOP repeatedly unless a condition test within the loop (such as EXITIF/THEN/ENDEXIT, or IF/THEN) transfers execution outside of the loop.

Parameters:

statement(s) One or more procedure lines that execute within the loop.

For more information, see LOOP/ENDLOOP.

ENDWHILE **Close WHILE statement**

Syntax: **WHILE *condition* DO**
proclines
ENDWHILE

Function: Forms the bottom of a WHILE loop. WHILE causes the procedure lines between DO and ENDWHILE to execute as long as the result of the expression following WHILE is true. Because the loop is tested at the top, the lines within the loop are never executed unless the expression is true.

Parameters:

condition	A Boolean expression (produces results of True or False).
proclines	Are program lines to execute if the expression is true.

For more information, see WHILE/DO/ENDWHILE.

EOF Test for end-of-file

Syntax: EOF(*path*)

Function: Tests for the end of a disk file. The function returns a value of True when it encounters an end-of-file; otherwise, it returns False. Use EOF with a READ or GET statement.

Parameters:

path	The number of the path you are accessing. BASIC09 automatically stores a path number into the variable you specify during a CREATE or OPEN operation.

Examples:

```
IF EOF(#PATH) THEN
CLOSE #PATH
ENDIF
```

Sample Program:

This procedure redirects a listing of the current directory into a file named Dirfile. It then lists Dirfile to the screen. EOF tells the WHILE/ENDWHILE loop when the READ operation reaches the end of the file.

```
PROCEDURE readfile
□DIM A:STRING[80]
□DIM PATH:BYTE
□SHELL "DIR > dirfile"
□OPEN #PATH,"dirfile":READ
□WHILE NOT EOF(#PATH) DO
□READ #PATH,A
□PRINT A
□ENDWHILE
□CLOSE #PATH
□END
```

ERR Return error code

Syntax: ERR

Function: Returns the error code of the most recent error. BASIC09 automatically sets the ERR code to zero after you reference it. ERR is only useful when used in conjunction with BASIC09's ON ERROR error trapping functions.

See Appendix A for a list of all BASIC09 error codes.

Parameters: None

Examples:

```
ERRNUM = ERR
IF ERRNUM = 218 THEN
PRINT "File already exists. Please use another
filename."
ENDIF
```

Sample Program:

This procedure displays the contents of a file you select. If the file doesn't exist (Error 216, Pathname not found), the procedure uses ERR to tell you. If an error other than Error 216 occurs, the procedure displays `I can't handle error` *xx*, where *xx* is the code of the error.

```
PROCEDURE readfile
□DIM READFILE:STRING; A:STRING[80]; PATH:BYTE
10□INPUT "Type the pathlist of the file to read...",READFILE
□ON ERROR GOTO 100  (* if an error occurs, skip to line 100.
□OPEN #PATH,READFILE:READ
□WHILE EOF(#PATH)<>TRUE DO
□READ #PATH,A
□PRINT A
□ENDWHILE
□CLOSE #PATH
□END
100□ERRNUM=ERR (* store the error code in ERRNUM.
□IF ERRNUM=216 THEN (* if file doesn't exist say so.
□PRINT "I can't find the file...Please try again."
□ON ERROR
□GOTO 10
□ENDIF
□PRINT "Sorry, I can't handle error number "; ERRNUM (* other error.
□CLOSE #PATH
□END
```

ERROR Simulate an error

Syntax: **ERROR *code***

Function: Simulates the error specified by *code*. You would mainly use this command to test ON ERROR GOTO routines. When BASIC09 encounters an ERROR statement, it proceeds as if the error corresponding to the specified code has occurred. Refer to Appendix A for a listing of error codes and their meanings.

Parameters:

code The code of the error you want to simulate.

Examples:

```
ERROR 207
ERRNUM = ERR
IF ERRNUM = 207 THEN
PRINT "Memory is full. The current data is being
saved to disk."
ENDIF
```

Sample Program:

This program creates a file named Test1. Before creating the file, it checks to see if it already exists. If the file exists, the procedure deletes it. An error trap catches any error that might occur. To test if the trap works for Error 216, "Pathname not found", the statement ERROR 216 is inserted as the fourth line. After testing the trap to make sure it works, delete this line to use the procedure.

```
PROCEDURE errortest
□DIM PATH,ERRNUM:BYTE; RESPONSE:STRING[1]
□BASE 0
□ON ERROR GOTO 10     (* set error trap
□ERROR 216            (* simulate error
□DELETE "test1"
□GOTO 100
10□ERRNUM=ERR
□IF ERRNUM=216 THEN
□INPUT "File doesn't exist...continue?
(Y/N)",RESPONSE
□IF RESPONSE="N" THEN
□END "Procedure terminated at your request..."
□ENDIF
□ENDIF
□ON ERROR             (* turn off error trap.
100□CREATE #PATH,"test1":WRITE
□END
```

EXITIF/THEN/ENDEXIT

Exit from loop if a condition is true

Syntax: **EXITIF *condition* THEN**
statement
ENDEXIT

Function: Use these statements with loop constructions (particularly LOOP and ENDLOOP) to provide an exit for what is otherwise an endless loop. EXITIF performs a test of a Boolean expression, such as A<B. The THEN statement precedes any operation you want to execute if the expression is true. You must always follow EXITIF with an ENDEXIT.

If the Boolean expression following an EXITIF is false, execution of the program transfers to the statement immediately following the body of the loop (after the ENDEXIT statement). Otherwise, BASIC09 executes the statement(s) between EXITIF and ENDEXIT, then transfers control to the statement following the body of the loop.

You can also use EXITIF and ENDEXIT with types of loop constructions other than LOOP/ENDLOOP.

Parameters:

Boolean expression	A comparison operation that returns either True or False, such as A=B, A<B, or A=B=C.
statement	An operation to be performed if the Boolean expression tested by EXITIF is True, such as: `PRINT A is less than B.`

Examples:

```
LOOP
COUNT=COUNT+1
EXITIF COUNT>100 THEN
DONE = TRUE
ENDEXIT
PRINT COUNT
X = COUNT/2
ENDLOOP
```

Sample Program:

This procedure simulates a gambling machine by randomly selecting among several fruit names and displaying them. It gives you a starting stake of $25 and, depending on the combination of fruit selected, it adds or subtracts from your stake.

If your stake drops to zero, an EXITIF statement ends the procedure and tells you that you're broke.

```
PROCEDURE onearm
□DIM FRUIT1,FRUIT2,FRUIT3,STAKE:INTEGER; FRUIT(8):
STRING[6]
□STAKE=25
□PRINT  \ PRINT "You have $"; STAKE; " to play
with."
□FOR T=1 TO 8
□READ FRUIT(T)
□NEXT T
□LOOP
□FRUIT1=RND(7)+1 \FRUIT2=RND(7)+1 \FRUIT3=RND(7)+1
□PRINT FRUIT(FRUIT1); " "; FRUIT(FRUIT2); " ";
FRUIT(FRUIT3)
□IF FRUIT(FRUIT1)=FRUIT(FRUIT2) AND FRUIT(FRUIT1)=
FRUIT(FRUIT3) THEN STAKE=STAKE+10
□ELSE
□IF FRUIT(FRUIT1)=FRUIT(FRUIT2) OR FRUIT(FRUIT1)=
FRUIT(FRUIT3) OR
□FRUIT(FRUIT2)=FRUIT(FRUIT3) THEN
□STAKE=STAKE+1
□ELSE
□STAKE=STAKE-1
□ENDIF
□ENDIF
```

```
□REM exit play loop is stake is less than $1.
□EXITIF STAKE<1 THEN
□PRINT
□PRINT "You're Busted...Better go home."
□ENDEXIT
□PRINT "Your stake is now $"; STAKE; "."
□PRINT
□PRINT
□INPUT "Press ENTER to pull again...",Z$
□ENDLOOP
□END
□DATA "ORANGE","APPLE","CHERRY","LEMON","BANANA"
□DATA "PEAR","PLUM","PEACH"
```

EXP Return natural exponent

Syntax: **EXP(*number*)**

Function: Returns the natural exponent of *number*, that is, e (2.71828183) to the power of *number*. *Number* must be positive.

This function is the inverse of the LOG function. Therefore, *number* = EXP(LOG(*number*)).

Parameters:

number A positive value.

Examples:

```
PRINT EXP(2)
```

Sample Program:

This procedure calculates the exponent of values in the range 0-1.

```
PROCEDURE exprint
□FOR T=0 TO 1 STEP .03
□PRINT EXP(T),EXP(T+.01),EXP(T+.02)
□NEXT T
□END
```

FALSE Assign Boolean value

Syntax: *variable* = FALSE

Function: FALSE is a Boolean function that always returns False. You can use FALSE and TRUE to assign values to Boolean variables.

Parameters: None

Examples:

```
DIM TEST:BOOLEAN
TEST=FALSE
```

Sample Program:

The procedure uses a Boolean variable to store True or False, depending on whether you answer some questions correctly or incorrectly.

```
PROCEDURE quiz
□DIM REPLY,VALUE:BOOLEAN; ANSWER:STRING[1];
QUESTION:STRING[80]
□FOR T=1 TO 5
□READ QUESTION,VALUE
□PRINT QUESTION
□PRINT "(T) = TRUE□□□□□□□□(F) = FALSE"
□PRINT "Select T or F:□□";
□GET #1,ANSWER
□IF ANSWER="T" THEN
□REPLY=TRUE
□ELSE
□REPLY=FALSE
□ENDIF
□IF REPLY=VALUE THEN
□PRINT \ PRINT "That's Correct...Good Show!"
□ELSE
□PRINT "Sorry, you're wrong...Better Luck next
time."
□ENDIF
□PRINT \ PRINT \ PRINT
```

```
NEXT T
DATA "In computer talk, CPU stands for Central
Packaging Unit.", FALSE
DATA "The actual value of 64K is 65536
bytes.",TRUE
DATA "The bits in a byte are normally numbered 0
through 7?",TRUE
DATA "BASIC09 has four data types.",FALSE
DATA "The LAND function is a Boolean type
operator.",FALSE
END
```

FIX Round a real number

Syntax: **FIX(*value*)**

Function: Rounds a real number to the nearest whole number and converts it to an integer-type number. Fix performs a function that is the opposite of the FLOAT function.

Parameters:

value Any real number.

Examples:

```
A=RND(10)
PRINT FIX(A)
```

Sample Program:

This procedure displays the FIXed values of seven constants.

```
PROCEDURE printfix
 PRINT FIX(1.2)
 PRINT FIX(1.3)
 PRINT FIX(1.5)
 PRINT FIX(1.8)
 PRINT FIX(99.566666)
 PRINT FIX(50.1)
 PRINT FIX(.7654321)
 PRINT FIX(-12.44)
 PRINT FIX(-9.99)
 END
```

FLOAT Convert from integer or byte to real

Syntax: **FLOAT(*value*)**

Function: Converts an integer- or byte-type value to real type. FLOAT performs a function that is the opposite of the FIX function.

Parameters:

value An integer- or byte-type number.

Examples:

```
DIM TEST:INTEGER
TEST=44
PRINT FLOAT(TEST)/3
```

Sample Program:

This procedure uses FLOAT to produce a real number result of an inch to centimeter conversion.

```
PROCEDURE convert
□DIM T:INTEGER; MEASURE:STRING[11]
□FOR T=1 TO 10
□IF T=1 THEN
□MEASURE="centimeter "
□ELSE
□MEASURE="centimeters"
□ENDIF
□PRINT T; " "; MEASURE; " is "; FLOAT(T)*.3937;
" inches."
□NEXT T
□END
```

FOR/NEXT/STEP Establish a loop

Syntax:
FOR *variable* = *init val* TO *end val* [STEP *value*]
[*procedure statements*]
NEXT *variable*

Function: Establishes a procedure loop that lets BASIC09 execute one or more procedure statements a specified number of times. The variables you use can be either integer or real type and can be negative, positive, or both. Loops using integer values execute faster than loops using real values.

BASIC09 executes the lines following the FOR statement until it encounters a NEXT statement. Then it either increases or decreases the initial value by one (the default) or by the value given STEP.

Parameters:

variable	Any legal numeric variable name.
init val	Any numeric constant or variable.
end val	Any numeric constant or variable.
value	Any numeric constant or variable.
procedure statements	Procedure lines you want to be executed within the loop.

Notes:

- If you provide an initial value that is greater than the final value, BASIC09 skips the program loop entirely unless you specify a negative STEP value. Specifying a negative value for STEP causes the loop to decrement from the initial value to the end value.

- When execution reaches the NEXT statement in a positive stepping loop, and the step value is less than or equal to the end value, BASIC09 branches back to the line after FOR and repeats the process. When the step value is greater than the end value, BASIC09 transfers execution to the statement following the NEXT statement.
- When execution reaches the NEXT statement in a negative stepping loop, and the step value is greater than or equal to the end value, BASIC09 branches back to the line after FOR and repeats the process. When the step value is less than the end value, execution continues following the NEXT statement.

Examples:

```
FOR COUNTER = 1 to 100 STEP .5
PRINT COUNTER
NEXT COUNTER

FOR X = 10 TO 1 STEP -1
PRINT X
NEXT X

FOR TEST = A TO B STEP RATE
PRINT TEST
NEXT TEST
```

Sample Program:

This procedure uses two *nested* FOR/NEXT loops to produce a multiplication table.

```
PROCEDURE multable
□PRINT USING "S45^","MULTIPLICATION TABLE"
□PRINT
□DIM I,J:INTEGER
□FOR I=1 TO 9
□FOR J=1 TO 9
□IF J>1 THEN
□PRINT I*J; TAB(5*J);
□ELSE PRINT I*J; "| ";
□ENDIF
□NEXT J
□IF I=1 THEN
□PRINT ""
```

```
□PRINT "-------------------------------------
---";
□ENDIF
□PRINT
□NEXT I
□END
```

GET Read a direct-access file record

Syntax: GET ***#path,varname***

Function: Reads a fixed-size binary data record from a file or device. Use GET to retrieve data from random access files.

Although you usually use GET with files, you can also use it to receive data for any outputting device, such as a keyboard or another computer. By dimensioning a string variable to the length of input you want, you can use GET to read a specified number of keystrokes, then continue program execution without requiring [ENTER] to be pressed.

For information about storing data in random access files, see Chapter 8, "Disk Files." Also see PUT, SEEK, and SIZE.

Parameters:

path	A variable name you choose in which BASIC09 stores the number of the path it opens to the device you specify or one of the standard I/O paths (0, 1, or 2).
varname	The variable in which you want to store the data read by the GET statement.

Examples:

```
GET #PATH,DATA$

GET #1,RESPONSE$

GET #INPUT,INDEX(X)
```

Sample Program:

This procedure directs a directory listing to a file named Dirfile. GET then reads the file, one character at a time in order to determine which characters are valid filename characters. The procedure creates a file containing all the filenames in the directory.

```
PROCEDURE filenames
□DIM DIRECTORY,FILENAME:STRING; CHARACTER:STRING[1]; FILES(125):STRING[15];
PATH,COUNT,T:INTEGER
□COUNT=0
□FILENAME=""
□FOR T=1 TO 125 (* initialize array elements to null.
□FILES(T)=""
□NEXT T
□INPUT "Pathlist of directory to read...",DIRECTORY (* dir to copy.
□ON ERROR GOTO 10
□DELETE "dirfile" (* if dirfile already exists, delete it.
10□ON ERROR
□SHELL "DIR "+DIRECTORY+" > dirfile" (* copy directory into file.
□OPEN #PATH,"dirfile":READ (* open the file for reading.
□REPEAT
□REM Get characters from the file until the first carriage return - the
beginning of the first filename.
□GET #PATH,CHARACTER (* get characters from the file.
□UNTIL CHARACTER=CHR$(13)
□REM
20□LOOP
□EXITIF EOF(#PATH) THEN
□GOTO 200 (* quit when end of file.
□ENDEXIT
□REM get a character from the file until it finds a non-valid filename
character.
□GET #PATH,CHARACTER
□REM
□EXITIF CHARACTER<=" " OR CHARACTER>"z" THEN
□GOTO 100
□ENDEXIT
□FILENAME=FILENAME+CHARACTER (* build the filename.
□ENDLOOP
100□WHILE NOT(EOF(#PATH)) DO
□GET #PATH,CHARACTER (* check for non-valid filename characters.
□EXITIF CHARACTER>" " AND CHARACTER<="z" THEN (* check if valid char.
□COUNT=COUNT+1
□FILES(COUNT)=FILENAME (* store filename in array.
□PRINT FILENAME, (* display the extracted filename.
□FILENAME="" (* set variable to NULL.
□FILENAME=FILENAME+CHARACTER (* last character begins new filename.
□GOTO 20 (* go get the rest of filename.
□ENDEXIT
□ENDWHILE
200□CLOSE #PATH
```

```
□DELETE "dirfile" (* names are all in array so delete file.
□CREATE #PATH,"dirfile":WRITE (* create the file again.
□FOR T=1 TO COUNT
□WRITE #PATH,FILES(T) (* fill the file with individual filenames.
□NEXT T
□CLOSE #PATH
□PRINT
□PRINT "□□□□□□□*The directory has "; COUNT; " entries"
□PRINT□"□□□□□□□□They are now stored in a file named Dirfile."
□END
```

GOSUB/RETURN

Jump to subroutine/ Return from subroutine

Syntax: GOSUB ***linenumber***

Function: Branches program execution to the specified line number.

BASIC09 lets you write programs with line numbers or without. You can also mix numbered and un-numbered lines within a single procedure. This means that, to use GOSUB, you need to number only the first line of the subroutine to which you want to branch.

Every subroutine you access with GOSUB must contain a RETURN statement. You can call a subroutine in this manner as many times as you want. When BASIC09 encounters the RETURN, it transfers program execution to the line following the GOSUB statement.

You can precede GOSUB with a test statement, such as IF or WHEN, that makes branching conditional.

You can nest GOSUB statements to any depth, depending on your computer's free memory.

Parameters:

linenumber	The number of the line where procedure execution is to continue.

Examples:

```
GOSUB 100
```

Sample Program:

The following procedure asks you for two numbers and an operator. It determines the line to jump to by the position of the operator in a table. GOSUB sends the procedure to execute the proper routine. RETURN sends the execution back to the main routine. To quit, enter a negative value.

```
PROCEDURE calc
□DIM NUM1,NUM2:REAL; OP:STRING[1]; A:INTEGER
1□INPUT "NUMBER 1 ";NUM1
□IF NUM1<0 THEN
□END
□ENDIF
□INPUT "NUMBER 2 ";NUM2
□INPUT "OPERATOR ";OP
□A=SUBSTR(OP,"+-*/^")
□ON A GOSUB 10,20,30,40,50
□GOTO 1
10□PRINT NUM1+NUM2 \ RETURN
20□PRINT NUM1-NUM2 \ RETURN
30□PRINT NUM1*NUM2 \ RETURN
40□PRINT NUM1/NUM2 \ RETURN
50□PRINT NUM1 NUM2 \ RETURN
□END
```

IF/THEN/ELSE/ENDIF

Test a Boolean expression

Syntax: **IF *condition* THEN *linenumber***
[ELSE
secondary action
ENDIF]

IF *condition* THEN
action
[ELSE
***secondary action*]**
ENDIF

Function: Tests a Boolean expression and executes *action* if the expression is true. Optionally, the statements execute a secondary action if the expression is not true. Each IF statement must be accompanied by THEN. If *action* is a line number, you can omit the ENDIF statement. For instance, both of the following statements operate in the same manner:

```
IF T=5 THEN 10
```

```
IF T=5 THEN
GOTO 10
ENDIF
```

Parameters:

condition	A Boolean expression (produces True or False).
linenumber	A line to which the procedure is to transfer execution if *condition* is true.
action	One or more procedure statements to be executed if *condition* is true.
secondary action	One or more procedure statements to execute if *condition* is false.

Examples:

```
IF A>B THEN 100

IF A<B THEN 100
ELSE
A=A-1
ENDIF

IF TEST=TRUE THEN
PRINT "The test is a success..."
ENDIF

IF A < B THEN
PRINT "A is less than B"
ELSE
PRINT "B is less than A"
ENDIF
```

Sample Program:

The following procedure is a *purge* procedure. Use it only with the GET Sample Program to delete one or more files from your current directory.

The Filenames procedure (see GET) stores the current directory's filenames in Dirfile. This procedure reads Dirfile, displays all the filenames, then asks you for a *wildcard*. Type in characters that identify a group of files you want to delete. The program deletes **all** files that contain, in the same order and case, the characters you type.

For instance, if you have four files named Test, File1, File2, and File3, and you type a wildcard of "File," the procedure deletes File1, File2, and File3, but does not delete Test. Delete all of the files in a directory by typing "*" as the wildcard.

Use this program carefully. Be sure you are in the right directory and that the wildcard characters you type are not contained in filenames other than the ones you want to delete. You might want to add a prompt to the procedure that lets you confirm each deletion before it happens.

```
PROCEDURE purge
□DIM PATH:INTEGER
□DIM NAME(100):STRING
□DIM WILDCARD:STRING
□X=0
□OPEN #PATH,"dirfile":READ
□WHILE NOT(EOF(#PATH)) DO
□X=X+1
□READ #PATH,NAME(X)
□ENDWHILE
□FOR T=1 TO X
□PRINT NAME(T),
□NEXT T
□INPUT "Wildcard Characters...",WILDCARD
□FOR T=1 TO X
□ON ERROR GOTO 100
□IF SUBSTR(WILDCARD,NAME(T))>0 OR WILDCARD="*"
THEN
□PRINT "DELETING "; NAME(T); " ......"
□DELETE NAME(T)
□ENDIF
10□NEXT T
□END
100□PRINT "* * * ERROR * * * "; NAME(T); " cannot
be deleted..continuing."
□GOTO 10
```

INKEY **Read a keypress**

Syntax: **RUN INKEY(*string*)**

Function: Reads a keypress, and stores the character of the key in the specified string variable.

Parameters:

string is a string variable into which INKEY stores the character you press.

Examples:

```
DIM CHAR:STRING[1]
CHAR=""
WHILE CHAR="" DO
RUN INKEY(CHAR)
ENDWHILE
PRINT ASC(CHAR)
```

Sample Program:

```
PROCEDURE Calculate
□DIM CHAR:STRING[1]
□DIM LOOKUP:STRING[7]
□DIM FIRST,SECOND:REAL
□DIM FLAG:INTEGER
□LOOKUP="+-*/^<>"
1 FLAG=0 \CHAR=""
□PRINT "Enter the first number to evaluate...";
□INPUT FIRST
□IF FIRST=0 THEN
□GOTO 100
□ENDIF
□PRINT "Enter the second number to evaluate...";
□INPUT SECOND
□PRINT "Press the key of the operator you want to
use..."
□PRINT " + - * / ^ < > ...";
□WHILE CHAR="" DO
□RUN INKEY(CHAR)
□ENDWHILE
□PRINT
□FLAG=SUBSTR(CHAR,LOOKUP)
□ON FLAG GOTO 10,20,30,40,50,60,70
10 PRINT FIRST+SECOND \ GOTO 1
20 PRINT FIRST-SECOND \ GOTO 1
30 PRINT FIRST*SECOND \ GOTO 1
40 PRINT FIRST/SECOND \ GOTO 1
50 PRINT FIRST^SECOND \ GOTO 1
60 PRINT FIRST<SECOND \ GOTO 1
70 PRINT FIRST>SECOND \ GOTO 1
100 PRINT "Procedure Terminated Due to 0
Input..."
□END
```

INPUT Get data from a device path

Syntax: **INPUT [#*path*,] [*prompt*,] *variable* [,*variable*...]**

Function: INPUT accepts input from the specified path. (The default is the keyboard.) When a procedure encounters INPUT, it displays a question mark and awaits data from the specified path. If you provide a string type prompt for INPUT, it displays the text of the prompt, rather than a question mark.

INPUT stores the data it collects in the variable you specify. The type of the receiving variable must match the type of data received.

Because INPUT sends data (the question mark prompt or the user-specified string prompt), it is really both an input and an output statement. This means that, if you use a path other than the standard input path, you should not use the UPDATE mode. If you do, the prompts produced by INPUT write to the file specified by the path number.

If the data received does not match the type of data INPUT expects, it displays the message:

```
**INPUT ERROR - RETYPE**
```

followed by a new prompt. You must then enter the entire input line, of the correct type, to satisfy INPUT. For more information, see GET.

Parameters:

path	Either a variable containing the path number, or the absolute path number to the file or device from which you want to receive input. If you want to receive input from the keyboard, do not include a path number.
prompt	Text you type as a message to be displayed when BASIC09 executes an INPUT statement.
variable	The variable name in which you want to store the data received by INPUT. The type of variable must match the type of input.

Examples:

```
INPUT NUMBER,NAME$,LOCATION

INPUT #PATH,X,Y,Z

INPUT "What is your selection: ";CHOICE

INPUT #HOST,"What's your ID number? ",IDNUM
```

Sample Program:

This procedure calculates the day of the week for a specified date. It asks you for the date using the INPUT command.

```
PROCEDURE weekday
□DIM X,Y,D,M,CALC:INTEGER; DAY,MONTH:STRING[2];
YEAR:STRING[4]; WEEKDAY (7):STRING[9]
□DIM ANUM,BNUM,CNUM,DNUM,ENUM,FNUM,GNUM,HNUM,
INUM:INTEGER
□PRINT USING "S80 ","Day of the Week Program","For
any year after 1752"
□PRINT
□PRINT "Enter day (e.g. 08): ";  \ INPUT DAY
□PRINT "  Enter month (e.g. 12): ";  \ INPUT MONTH
□PRINT "  Enter year (e.g. 1986): ";  \ INPUT YEAR
□Y=VAL(YEAR) \M=VAL(MONTH) \D=VAL(DAY)
□FOR X=1 TO 7
□READ WEEKDAY(X)
□NEXT X
□ANUM=INT(.6+1/M)
□BNUM=Y-ANUM
```

```
□CNUM=M+12*ANUM
□DNUM=BNUM/100
□ENUM=INT(DNUM/4)
□FNUM=INT(DNUM)
□GNUM=INT(5*BNUM/4)
□HNUM=INT(13*(CNUM+1)/5)
□INUM=HNUM+GNUM-FNUM+ENUM+D-1
□INUM=INUM-7*INT(INUM/7)+1
□PRINT
□PRINT "The day of the week on "; M; "/"; D;
"/"; Y; " is..."; WEEKDAY (INUM)
□DATA "Sunday","Monday","Tuesday","Wednesday",
"Thursday","Friday","Saturday"
□END
```

INT Convert real number to whole number

Syntax: INT(*value*)

Function: Converts a real number to a whole number by truncating any fractional part of the real number.

Parameters:

value Any negative or positive real number.

Examples:

```
PRINT INT(77.89)

PRINT INT(NUM)

PRINT INT(-8.12)
```

Sample Program:

The RND function produces real numbers. This procedure uses INT to convert the real RND output to integer values.

```
PROCEDURE integer
□DIM T:INTEGER
□FOR T=1 TO 10
□R=RND(50)-25
□PRINT R,INT(R)
□NEXT T
□END
```

KILL Remove a procedure from memory

Syntax: KILL ***procedure***

Function: Unlinks (removes) an external procedure from the BASIC09 procedure directory. If the procedure is not external, but resides in BASIC09's workspace, KILL has no effect.

Use KILL to remove auto-loaded (packed) procedures that are called by RUN or CHAIN. You can also use KILL with auto-loading procedures as a method to overlay programs within BASIC09.

> **Warning:** Be certain you do not KILL an active procedure. Also be certain that when you use RUN and KILL together, that both statements use the same string variable that contains the name of the procedure to RUN and KILL.

Parameters:

procedure	The name of the external procedure you want to KILL. *Procedure* can either be a name or a variable containing the procedure name.

Examples:

```
PROCEDURENAME$ = "AVERAGE"
RUN PROCEDURENAME$
KILL PROCEDURENAME$
```

```
INPUT "Which test do you want to run? ",TEST$
RUN TEST$
KILL TEST$
```

Sample Program:

This procedure calls a procedure named Show to display ASCII values on the screen. When it no longer needs the Show procedure, it removes Show from memory using KILL.

```
PROCEDURE produce
□DIM T,U:INTEGER
□DIM NUM,NUM1,NUM2,TABLE,PROCNAME:STRING
□PROCNAME=SHOW
□TABLE="123456789ABCDEF"
□FOR T=8 TO 15
□FOR U=1 TO 15
□NUM1=MID$(TABLE,T,1)
□NUM2=MID$(TABLE,U,1)
□NUM=NUM1+NUM2    (* parameter to pass to Show.
□RUN PROCNAME(NUM)
□NEXT U
□NEXT T
□KILL "PROCNAME" (* remove Show from the workspace.
□END

PROCEDURE SHOW
□PARAM NUM:STRING
□SHELL "DISPLAY "+NUM
□END
```

LAND Returns the logical AND of two numbers

Syntax: LAND(***num1,num2***)

Function: Performs the logical AND function on a byte- or integer-type value. The operation involves a bit-by-bit logical AND of the two numbers you specify. For instance, if you LAND 5 and 6, the logic is like this:

Decimal 5 = Binary 0101
Decimal 6 = Binary 0110

```
      0101
AND   0110
----------
=     0100  = 4 Decimal
```

Parameters:

num1 A byte- or integer-type number.

num2 A byte- or integer-type number.

Examples:

```
PRINT LAND(11,12)

PRINT LAND($20,$FF)
```

Sample Program:

The following procedure asks eight questions and uses the eight bits of one byte (contained in the variable STORAGE) to indicate either a "yes" or "no" answer. If the answer is "yes," it sets a corresponding bit to 1. If the answer is "no," it sets a corresponding bit to 0, using LAND. This procedure operates in conjunction with the sample program for LXOR.

```
PROCEDURE questions
□DIM QUESTION:STRING[60]; T:INTEGER; X,STORAGE:BYTE
□DIM ANSWER:STRING[1]
□X=1
□FOR T=1 TO 8
□READ QUESTION
□PRINT QUESTION; " (Y/N)? ";
□GET #0,ANSWER
□PRINT
□IF ANSWER="y" OR ANSWER="Y" THEN
□STORAGE=LOR(STORAGE,X) (* OR STORAGE if yes.
□ELSE
□STORAGE=LAND(STORAGE,LNOT(X)) (* LAND STORAGE with NOT value if no.
□ENDIF
□X=X*2
□NEXT T
□RUN summary(STORAGE)
□END
□DATA "Do you have more than one Color Computer"
□DATA "Do you use your Color Computer for games"
□DATA "Do you use your Color Computer for word processing"
□DATA "Do you use your Color Computer for business applications"
□DATA "Do you use your Color Computer at home"
□DATA "Do you use your Color Computer at the office"
□DATA "Do you use your Color Computer more than two hours a day"
□DATA "Do you share your Color Computer with others"
```

LEFT **Returns characters from the left portion of a string**

Syntax: **LEFT$(*string,length*)**

Function: Returns the specified number of characters from the specified string, beginning at the leftmost character. If *length* is the same as or greater than the number of characters in *string*, then LEFT$ returns all the characters in the string.

Parameters:

string	A sequence of ASCII characters or a string variable name.
length	The number of characters you want to access.

Examples:

```
PRINT LEFT$("HOTDOG",3)

PRINT LEFT$(A$,6)
```

Sample Program:

The following procedure extracts the first name from a list of ten names with the LEFT$ function.

```
PROCEDURE firstname
□DIM NAMES:STRING; FIRSTNAME:STRING[10]
□PRINT "Here are the first names:"
□FOR T=1 TO 10
□READ NAMES
□POINTER=SUBSTR(" ",NAMES) (* find space between first and last names.
□FIRSTNAME=LEFT$(NAMES,POINTER-1) (* extract first name.
□PRINT FIRSTNAME (* print first name.
□NEXT T
□END
□DATA "Joe Blonski","Mike Marvel","Hal Skeemish","Fred Laungly"
□DATA "Jane Mistey","Wendy Paston","Martha Upshong","Jacqueline Rivers"
□DATA "Susy Reetmore","Wilson Creding"
```

LEN Returns the length of a string

Syntax: LEN(*string*)

Function: Returns the number of characters in a string. Counts blanks or spaces as characters.

Parameters:

string A literal string or a variable containing string characters.

Examples:

```
PRINT LEN("ABCDEFGHIJKLM")

PRINT LEN(NAME$)

NAME$ = "JOE"
ADDRESS$ = "2244 LANCASTER"
TOTALLEN = LEN(NAME$)+LEN(ADDRESS$)
```

Sample Program:

The following procedure uses LEN to determine which name in a list is longest.

```
PROCEDURE longname
□DIM NAMES,LNAME:STRING; LONGEST,LENGTH:INTEGER
□NAMES="" \LNAME="" \LENGTH=0 \LONGEST=0
□FOR T=1 TO 10
□READ NAMES
□LENGTH=LEN(NAMES)
□IF LONGEST<LENGTH THEN
□LONGEST=LENGTH
□LNAME=NAMES
□ENDIF
□NEXT T
□PRINT "The longest name is "; LNAME; " with "; LONGEST; " characters."
□END
□DATA "Joe Blonski","Mike Marvel","Hal Skeemish","Fred Laungly"
□DATA "Jane Misty","Wendy Paston","Martha Upshong","Jacqueline Rivers"
□DATA "Susy Reetmore","Wilson Creding"
```

LET Assigns a variable's value

Syntax: [LET] ***variable = expression***

Function: Assigns a value to a variable. BASIC09 does not require the LET statement to assign values but does accept it in order to be compatible with versions of BASIC that do require it.

Parameters:

variable	The variable to which you want to assign a value.
expression	Either a numeric or string constant or a numeric or string expression.

Notes:

- The result of the LET expression must be of the same type as, or compatible with, the variable in which it is stored.
- BASIC09's assignment function accepts either = or := as assignment operators. The := form helps to distinguish assignment operations from comparisons (test for equality) and is compatible with Pascal programming.
- Use BASIC09's assignment function to copy entire arrays or complex data structures to another array or complex data structure. The data structures do not need to be of the same type or *shape*, but the size of the destination structure must be the same as or larger than the source structure. This means the assignment function can perform unusual type conversions. For example, you can copy a string variable of 80 characters into a one-dimensional array of 80 bytes.

Examples:

```
LET A = 5

LET A := B

ANSWER = A * B

LET NAME$ := "JOE"

NAME $ = FIRSTNAME$ + " " + LASTNAME$
```

Sample Program:

This procedure uses LET to assign a random value to the variable R.

```
PROCEDURE getint
□DIM T:INTEGER
□FOR T=1 TO 10
□LET R=RND(50)-25
□PRINT R,INT(R)
□NEXT T
□END
```

LNOT Performs a logical NOT on a number

Syntax: LNOT(*value*)

Function: Performs the logical NOT function on an integer or byte type number. The operation involves a bit-by-bit logical complement operation of the number you specify. For instance, if *value* is 188, the logic looks like this:

188 Decimal = 10111100 Binary

NOT 10111100
= 01000011

01000011 Binary = 67 Decimal

LNOT changes each bit in a binary number to its complementary binary value—all 1 values become 0 and all 0 values become 1. LNOT returns an integer result; it is not a Boolean operator.

Parameters:

value	Any decimal or hexadecimal integer or byte number. Precede hexadecimal numbers with $.

Examples:

```
PRINT LNOT(88)

A = LNOT(B)

A = LNOT($44)
```

Sample Program:

This procedure uses one byte (contained in the variable STORAGE) to indicate the results of eight questions. Each bit in the byte indicates a Yes or No answer (Yes = 1 and No = 0). The combination logic of LAND and LNOT masks the byte X so that it affects only the appropriate bit of STORAGE to set it to 0 if the answer is No. LOR sets the appropriate bit to 1 if the answer is Yes. The procedure operates in conjunction with the LXOR sample program.

```
PROCEDURE questions
□DIM QUESTION:STRING[60]; T:INTEGER; X,STORAGE:BYTE
□DIM ANSWER:STRING[1]
□X=1
□FOR T=1 TO 8
□READ QUESTION
□PRINT QUESTION; " (Y/N)? ";
□GET #0,ANSWER
□PRINT
□IF ANSWER="y" OR ANSWER="Y" THEN
□STORAGE=LOR(STORAGE,X) (* Answer is yes, set bit to 1.
□ELSE
□STORAGE=LAND(STORAGE,LNOT(X)) (* Answer is no, set bit to 0.
□ENDIF
□X=X*2
□NEXT T
□PRINT STORAGE
□RUN summary(STORAGE)
□END
□DATA "Do you have more than one Color Computer"
□DATA "Do you use your Color Computer for games"
□DATA "Do you use your Color Computer for word processing"
□DATA "Do you use your Color Computer for business applications"
□DATA "Do you use your Color Computer at home"
□DATA "Do you use your Color Computer at the office"
□DATA "Do you use your Color Computer more than two hours a day"
□DATA "Do you share your Color Computer with others"
```

LOG Returns natural logarithm

Syntax: LOG(***number***)

Function: Computes the natural logarithm of a number that is greater than zero. BASIC09 returns the logarithm as a real type result.

Parameters:

number	Any integer, byte, or real number.

Examples:

```
PRINT LOG(3.14159)

LOGVALUE = LOG(88/PI)
```

Sample Program:

This procedure calculates the natural log and the log to base 10 of the values 1-7.

```
PROCEDURE logs
□DIM NUM,T:INTEGER
□FOR T=1 TO 7
□PRINT "The LOG of "; T; " to the natural base =
"; LOG(T)
□PRINT "The LOG of "; T; " to base 10 = ";
LOG10(T)
□PRINT
□NEXT T
□END
```

LOG10 Returns base 10 logarithm

Syntax: **LOG10(*number*)**

Function: Calculates the base 10 logarithm of a number. BASIC09 returns the logarithm as a real number.

Parameters:

number Any byte, integer, or real value.

Examples:

```
PRINT LOG10($45)

PRINT LOG10(A)

PRINT LOG10(A/12)
```

Sample Program:

This procedure calculates the natural log and the log to base 10 of the values 1-7.

```
PROCEDURE logs
□DIM NUM,T:INTEGER
□FOR T=1 TO 7
□PRINT "The LOG of "; T; " to the natural base =
"; LOG(T)
□PRINT "The LOG of "; T; " to base 10 = ";
LOG10(T)
□PRINT
□NEXT T
□END
```

LOOP/ENDLOOP
Establishes/Closes a loop

Syntax: **LOOP**
statement(s)
ENDLOOP

Function: Establishes a loop in which you can install EXITIF tests at any location. The LOOP and ENDLOOP statements define the body of the loop. EXITIF tests for a condition which, if TRUE, causes alternate actions, the transfer of procedure execution to another routine, or both.

If you do not include an EXITIF statement, the loop cannot terminate.

Parameters:

statement(s)	One or more procedure lines to execute within the loop.

Examples:

```
LOOP
COUNT = COUNT+1
EXITIF COUNT > 100 THEN
DONE = TRUE
ENDEXIT
PRINT COUNT
X=COUNT/2
ENDLOOP
```

```
INPUT X,Y
LOOP
PRINT
EXITIF X<0 THEN
PRINT "X became 0 first"
END
ENDEXIT
X = X-1
EXITIF Y=0 THEN
PRINT "Y became 0 first"
```

```
END
ENDEXIT
Y=Y-1
ENDLOOP
```

Sample Program:

This procedure simulates a gambling machine that awards cash returns depending on a random selection of kinds of fruits. You begin with a stake of $25 and win or lose according to random selections of the procedure.

The program uses LOOP/ENDLOOP to keep operating until you run out of cash.

```
PROCEDURE bandit
□DIM FRUIT1,FRUIT2,FRUIT3,STAKE:INTEGER;
FRUIT(10):STRING[6]
□STAKE=25
□PRINT  \ PRINT "You have $"; STAKE; " to play
with."
□FOR T=1 TO 10
□READ FRUIT(T)
□NEXT T
□LOOP
□FRUIT1=RND(9)+1 \FRUIT2=RND(9)+1 \FRUIT3=RND(9)+1
□PRINT FRUIT(FRUIT1); " "; FRUIT(FRUIT2); " ";
FRUIT(FRUIT3)
□IF FRUIT(FRUIT1)=FRUIT(FRUIT2) AND FRUIT(FRUIT1)=
FRUIT(FRUIT3) THEN
□STAKE=STAKE+10
□ELSE
□IF FRUIT(FRUIT1)=FRUIT(FRUIT2) OR FRUIT(FRUIT2)=
FRUIT(FRUIT3) THEN
□STAKE=STAKE+2
□ELSE
□IF FRUIT(FRUIT1)=FRUIT(FRUIT3) THEN
□STAKE=STAKE+1
□ELSE STAKE=STAKE-1
□ENDIF
□ENDIF
□ENDIF
□EXITIF STAKE<1 THEN
□PRINT
□PRINT "You're Busted...Better go home."
```

```
PROCEDURE questions
□DIM QUESTION:STRING[60]; T:INTEGER;
X,STORAGE:BYTE
□DIM ANSWER:STRING[1]
□X=1
□FOR T=1 TO 8
□READ QUESTION
□PRINT QUESTION; " (Y/N)? ";
□GET #0,ANSWER
□PRINT
□IF ANSWER="y" OR ANSWER="Y" THEN
□STORAGE=LOR(STORAGE,X)
□ELSE
□STORAGE=LAND(STORAGE,LNOT(X))
□ENDIF
□X=X*2
□NEXT T
□PRINT STORAGE
□RUN summary(STORAGE)
□END
□DATA "Do you have more than one Color Computer"
□DATA "Do you use your Color Computer for games"
□DATA "Do you use your Color Computer for word
processing"
□DATA "Do you use your Color Computer for business
applications"
□DATA "Do you use your Color Computer at home"
□DATA "Do you use your Color Computer at the
office"
□DATA "Do you use your Color Computer more than
two hours a day"
□DATA "Do you share your Color Computer with
others"
```

LXOR Returns logical XOR of two numbers

Syntax: **LXOR(*value1,value2*)**

Function: Performs the logical XOR function on two-byte, or integer-type, values. For instance, if you LXOR the numbers 5 and 6 the logic is like this:

Decimal 5 = Binary 0101
Decimal 6 = Binary 0110

```
      0101
LXOR  0110
-----------
=     0011  = 3 Decimal
```

If one bit or the other bit in the evaluation is 1, but not both, LXOR returns a result of 1. Otherwise, LXOR returns a result of 0.

Parameters:

value1	A byte or integer number.
value2	A byte or integer number.

Examples:

```
PRINT LXOR(11,12)

PRINT LXOR($20,$FF)
```

Sample Program:

The following program summarizes the results of the sample program for LOR. The LOR program stored the answers to eight questions in a single byte. This procedure reads the byte and displays appropriate comments. LXOR checks to see if two of the answers are "yes" or "no."

```
□ENDEXIT
□PRINT "Your stake is now $"; STAKE; "."
□PRINT
□PRINT
□INPUT "Press ENTER to pull again...",Z$
□ENDLOOP
□END
□DATA "ORANGE","APPLE","CHERRY","LEMON","BANANA"
□DATA "PEAR","PLUM","PEACH","GRAPE","APRICOT"
```

LOR Returns logical OR of two numbers

Syntax: LOR(***value1,value2***)

Function: Performs the logical OR function on a byte- or integer-type value. The operation involves a bit-by-bit logical OR operation on two values. For instance, if you LOR the numbers 5 and 6, the logic is like this:

Decimal 5 = Binary 0101
Decimal 6 = Binary 0110

```
      0101
OR    0110
----------
=     0111  = 7 Decimal
```

If one bit or the other bit is 1, LOR returns a result of 1. Otherwise, LOR returns a result of 0.

Parameters:

value1	A byte or integer number.
value2	A byte or integer number.

Examples:

```
PRINT LOR(11,12)

PRINT LOR($20,$FF)
```

Sample Program:

This procedure stores the answers to eight "yes" or "no" questions in one byte, named STORAGE. If you answer "yes" to a prompt, the procedure sets a corresponding bit to 1. If you answer "no" to a prompt, the procedure sets a corresponding bit to 0. The procedure uses LOR to set bits to 1 by *masking* all bits except the one it needs to set. The procedure operates in conjunction with the LXOR sample program.

```
PROCEDURE summary
□DIM T:INTEGER; A,B,X,TEST,TEST2:BYTE; SUMMARY:
STRING[50]
□PARAM STORAGE:BYTE
□A=0 \B=0
□PRINT  \ PRINT
□PRINT "The following is a summary of the
questionnaire answers:"
□PRINT
□PRINT "The surveyee: "
□X=1
□FOR T=1 TO 8
□TEST=LAND(STORAGE,X)
□READ SUMMARY
□IF TEST>0 THEN
□PRINT TAB(10); SUMMARY
□ENDIF
□X=X*2
□NEXT T
□IF LAND(STORAGE,128)>0 THEN
□A=1
□ENDIF
□IF LAND(STORAGE,64)>0 THEN
□B=1
□ENDIF
□TEST2=LXOR(A,B)
□IF TEST2=1 THEN
□PRINT "This computer owner either uses the
computer"
□PRINT "more than two hours a day or shares it
with others."
□PRINT "This is a heavy use situation."
□ENDIF
□TEST2=LAND(A,B)
□IF TEST2=1 THEN
□PRINT "This computer user uses the computer more
than two"
□PRINT "hours per day and shares it with others.
This is a"
□PRINT "super heavy use situation."
□ENDIF
□END
□DATA "Uses more than one computer"
□DATA "Plays games"
```

```
□DATA "Uses the computer for word processing"
□DATA "Uses the computer for business"
□DATA "Keeps a Color Computer at home"
□DATA "Keeps a Color Computer at the office"
□DATA "Uses the computer more than two hours a
day"
□DATA "Shares the computer with others"
```

MID$ Returns characters from within a string

Syntax: MID$(***string,begin,length***)

Function: Returns a substring *length* characters long, beginning at *begin*. Use MID$ to "take apart" a string consisting of a number of elements.

Parameters:

string	A sequence of string type characters or a string type variable.
begin	The position (an integer value) in *string* of the first character to retrieve.
length	The number of characters you want to retrieve.

Examples:

```
NAME$ = "JONES, JOHN M."
LASTNAME$ = MID$(NAME$,8,6)
FIRSTNAME$ = MID$(NAME$,1,5)
INITIAL$ = MID$(NAME$,15,2)
```

Sample Program:

This procedure reverses a word or phrase you type. MID$ reads each character in your phrase from the end to the beginning.

```
PROCEDURE reverse
□DIM PHRASE:STRING; T,BEGIN:INTEGER
□PRINT "Type a word or phrase you want to
reverse:";
□PRINT
□INPUT PHRASE
□BEGIN=LEN(PHRASE)
□PRINT "This is how your phrase looks backwards:"
□FOR T=BEGIN TO 1 STEP -1
□PRINT MID$(PHRASE,T,1);
□NEXT T
□PRINT
□END
```

MOD Returns modulus of a division

Syntax: MOD(***number1,number2***)

Function: Returns the modulus (remainder) of a division. MOD divides *number1* by *number2* and calculates the remainder. You can use MOD to put a limit on a numeric variable. For instance, regardless of the value of X, MOD(X,3) produces numbers only in the range 0 through 2. MOD(X,5) produces numbers only in the range of 0 through 4.

You can use MOD to cause repeating sequences. For instance, in a loop, MOD(X,3) produces a repeating sequence of 0, 1, 2, where X increases by 1 in each step of the loop.

Parameters:

number1	A byte, integer or real number dividend.
number2	A byte, integer or real number divisor.

Examples:

```
PRINT MOD(99,5)
```

Sample Program:

This procedure uses MOD to execute repeatedly routines that display asterisks on the screen. There are eight subroutines that the MOD function selects over and over through 100 passes.

```
PROCEDURE stardown
□DIM T:INTEGER
□SHELL "TMODE -PAUSE"
□FOR T=1 TO 100
□ON MOD(T,8)+1 GOSUB 10,20,30,40,50,60,70,80
□NEXT T
□SHELL "TMODE PAUSE"
□END
10□PRINT USING "S10^","*" \ RETURN
20□PRINT USING "S10^","**" \ RETURN
30□PRINT USING "S10^","***" \ RETURN
40□PRINT USING "S10^","****" \ RETURN
50□PRINT USING "S10^","*****" \ RETURN
60□PRINT USING "S10^","****" \ RETURN
70□PRINT USING "S10^","***" \ RETURN
80□PRINT USING "S10^","**" \ RETURN
□END
```

NEXT Causes repetition in a FOR loop

Syntax: **FOR *variable* = *init val* TO *end val* [STEP *value*]**
[*procedure statements*]
NEXT *variable*

Function: NEXT forms the bottom end of a FOR/NEXT loop. Any program statements between FOR and NEXT are executed once for each repetition of the loop, from the initial value to end value.

Parameters:

variable	Any legal numeric variable name.
init val	Any numeric constant or variable.
end val	Any numeric constant or variable.
value	Any numeric constant or variable.
procedure statements	Procedure lines you want to execute within the loop.

For more information, see FOR/NEXT/STEP.

NOT Returns the complement of a value

Syntax: NOT(***value***)

Function: Returns the logical complement of a Boolean value or expression.

Parameters:

value	A Boolean value (True or False), or an expression resulting in a Boolean value.

Examples:

```
DIM TEST:BOOLEAN
WHILE NOT(TEST) DO
A=A+1
TEST=A=B
ENDWHILE
```

Sample Program:

This procedure redirects the current directory listing to a file named Dirfile. It then opens Dirfile and reads the contents, displaying each line on the screen. It uses NOT in a WHILE/ENDWHILE loop to make sure that the end of the file has not been reached before trying to read another entry.

```
PROCEDURE readfile
□DIM A:STRING[80]
□DIM PATH:BYTE
□SHELL "DIR > dirfile"
□OPEN #PATH,"dirfile":READ
□WHILE NOT EOF(#PATH) DO
□READ #PATH,A
□PRINT A
□ENDWHILE
□CLOSE #PATH
□END
```

ON ERROR/GOTO

Establishes an error trap

Syntax: **ON ERROR [GOTO *linenum*]**

Function: Sets an error *trap* that transfers control to the specified line number in a procedure. This lets your program recover from an error and continue execution. To use these commands, your program must have at least one numbered line—the line to branch to in the event of an error.

Parameters:

linenum	The line to which you want BASIC09 to branch should an error occur.

Notes:

- ON ERROR GOTO is effective only with non-fatal, run-time errors. If such an error occurs without a preceding ON ERROR GOTO statement, BASIC09 enters the DEBUG mode. You must specify ON ERROR GOTO before an error occurs.
- You turn on error trapping by specifying ON ERROR GOTO *linenum*. You turn off error trapping by specifying ON ERROR without a line number.
- Use ON ERROR GOTO with the ERR function (that returns the code of the last error) to specify a particular action for a particular error. You can also use ERROR to simulate an error to test error trapping. For more information on this, see ERROR.

Examples:

```
□DIM FILENAME:STRING
□DIM PATH:INTEGER
10□INPUT "Name of file to create? ",FILENAME
□ON ERROR GOTO 100
□CREATE #PATH,FILENAME:UPDATE
□END
100□PRINT "That file already exists...please
choose another name..."
□GOTO 10
□END
```

Sample Program:

If you created a directory file with the GET sample program, you can use this procedure to delete files from the original directory using key characters. For instance, you might type XX as key characters. This means that any filename containing the character group XX is deleted. You can select any key characters you wish, but **be sure they apply only to files you want to delete.**

If you want to delete all the files in the directory, type an asterisk (*) when asked for key characters.

This procedure uses ON ERROR to let the procedure continue, even if a directory entry cannot be deleted—if an entry is a subdirectory. Without the ON ERROR function, the procedure would produce an error and cease execution when it tried to delete a subdirectory.

```
PROCEDURE purge
□REM Use caution with this procedure
□REM Be sure to specify key characters
□REM that exist only in the files you
□REM want to delete!

□DIM PATH:INTEGER
□DIM NAME(100):STRING
□DIM WILDCARD:STRING
□X=0
□OPEN #PATH,"dirfile":READ
□WHILE NOT(EOF(#PATH)) DO
□X=X+1
□READ #PATH,NAME(X)
```

```
□ENDWHILE
□FOR T=1 TO X
□PRINT NAME(T),
□NEXT T
□INPUT "Wildcard Characters...",WILDCARD
□FOR T=1 TO X
□ON ERROR GOTO 100
□IF SUBSTR(WILDCARD,NAME(T))>0 OR WILDCARD="*"
THEN
□PRINT "DELETING "; NAME(T); " ......"
□DELETE NAME(T)
□ENDIF
10□NEXT T
□END
100□PRINT "*□*□*□ERROR,□"; NAME(T); "□cannot be
deleted...continuing."
□GOTO 10
□END
```

ON/GOSUB Jumps to subroutine on a specified condition

Syntax: ON *pos* GOSUB *linenum* [,*linenum*,...]

Function: Transfers procedure control to the line number located at position *pos* in the list of line numbers immediately following the GOSUB command. For example, if *pos* equals 1, BASIC09 branches to the first line number it encounters in the list. If *pos* equals 2, BASIC09 branches to the second line number it encounters in the list. If *pos* is greater than the number of items in the list, execution continues with the next command line. To use ON/GOSUB you must have numbered lines to match the line numbers in your list. End the routines accessed by ON/GOSUB with a RETURN statement.

Parameters:

pos An integer value pointing to a line number in a list of line numbers.

linenum Any numbered line in the procedure.

Examples:

```
PRINT "You can now: (1) End the program (2) Print
the results"
PRINT "             (3) Try again       (4) Start
a new program"
INPUT "Type the letter of your choice: ",CHOICE
ON CHOICE GOSUB 100, 200, 300, 400
```

Sample Program:

This procedure uses MOD to execute repeatedly a sequence of GOSUB commands. A loop of index of 80 causes execution to jump to each line number in the list 10 times.

```
PROCEDURE repeat
□SHELL "TMODE -PAUSE"
□DIM T:INTEGER
□FOR T=1 TO 80
□ON MOD(T,8)+1 GOSUB 10,20,30,40,50,60,70,80
□NEXT T
□SHELL "TMODE PAUSE"
□END
10□PRINT USING "S10^","*" \ RETURN
20□PRINT USING "S10^","**" \ RETURN
30□PRINT USING "S10^","***" \ RETURN
40□PRINT USING "S10^","****" \ RETURN
50□PRINT USING "S10^","*****" \ RETURN
60□PRINT USING "S10^","****" \ RETURN
70□PRINT USING "S10^","***" \ RETURN
80□PRINT USING "S10^","**" \ RETURN
□END
```

ON/GOTO **Jump to line number on a specified condition**

Syntax: ON *pos* GOTO *linenum* [,*linenum*,...]

Function: Transfers procedure control to the line number located at position *pos* in the list of line numbers immediately following the GOTO command. For example, if *pos* equals 1, BASIC09 branches to the first line number it encounters in the list. If *pos* equals 2, BASIC09 branches to the second line number it encounters in the list. If *pos* is greater than the number of items in the list, execution continues with the next command line. To use ON/GOTO you must have numbered lines to match the line numbers in the list.

Parameters:

pos	An integer value in a range from 1 to the number of items in the list following GOTO.
linenum	Any numbered line in the procedure.

Examples:

```
PRINT "You can now: (1) End the program (2) Print
the results"
PRINT "               (3) Try again        (4) Start
a new program"
INPUT "Type the letter of your choice: ",choice
ON CHOICE GOTO 100, 200, 300, 400
```

Sample Program:

This procedure converts decimal numbers to binary. It uses ON GOTO to execute the operation you select from a menu: Convert a number, display the result of all conversions, or end the program.

```
PROCEDURE bicalc
□DIM□NUMBER,NUM,X,STORAGE:INTEGER;□BI:STRING;
□ARRAY(50,2):STRING
□COUNT=0
```

```
10□BI="" \NUMBER=0 \NUM=0 \X=0 \STORAGE=0
□INPUT "Number to convert to binary ",NUMBER
□IF NUMBER=0 THEN END
□ENDIF
□NUM=LOG10(NUMBER)/.3
□NUM=2^NUM \STORAGE=NUMBER
□REPEAT
□X=NUMBER/NUM
□IF X>0 THEN BI=BI+"1"
□NUMBER=MOD(NUMBER,NUM)
□ELSE BI=BI+"0"
□ENDIF
□NUM=NUM/2
□UNTIL NUM<=1
□IF NUMBER>0 THEN
□BI=BI+"1"
□ELSE^BI=BI+"0"
□ENDIF
□PRINT STORAGE; " = "; BI; " in binary."
□PRINT
□COUNT=COUNT+1
□ARRAY(COUNT,1)=STR$(STORAGE)
□ARRAY(COUNT,2)=BI
12□PRINT "Do you want to: (1) Convert another
number."
□PRINT "□□□□□□□□□□□□□□□□□□□□(2) Display all calculations
thus far."
□PRINT "□□□□□□□□□□□□□□□□□□□□(3) End the program."
□INPUT "Enter 1, 2, or 3...",choice
□ON choice GOTO 10,20,30
□END
20□FOR T=1 TO COUNT
□PRINT ARRAY(T,1); " = "; ARRAY(T,2)
□NEXT T
□GOTO 12
30□PRINT  \ PRINT " Program Terminated"
□END
```

OPEN Opens a path to a device

Syntax: **OPEN *#path,"pathlist"*** [***access mode***][+***access mode***][+...]

Function: Opens an input/output path to a disk file or to a device. When you open a file, you can select one or more of the following access modes:

Mode	Function
READ	Lets you read (receive) data from a file or device but does not allow you to write (send) data.
WRITE	Lets you write data to a file or device but does not allow you to read data.
UPDATE	Lets you both read from and write to a file or device.
EXEC	Specifies that the file you want to access is in the current execution directory.
DIR	Specifies that the file you want to access is a directory-type file.

Parameters:

path — The variable in which BASIC09 stores the number of the newly opened path.

pathlist — The route to the file or device to be opened, including the filename if appropriate.

access mode — The type of access the system is to allow for the file or device. Use a plus symbol to specify more than one type of access.

Notes:

- The access mode defines the direction of I/O transfers.
- Because OS-9 files are byte-addressed and are unformatted, you can set up the filing system you want for a particular application. Your system can read the data contained in a file as single bytes or in groups of any size you want.
- You can expand a file using PRINT, WRITE, or PUT statements to write beyond the current end-of-file.

Examples:

```
OPEN #TRANS,"transportation":UPDATE

OPEN #SPOOL,"/user4/report":WRITE

OPEN #OUTPATH,name$:UPDATE+EXEC
```

Sample Program:

This procedure opens a path to both the SYS directory on Drive /D0 and the error message file.

```
PROCEDURE readerr
□DIM A:STRING[80]
□DIM PATH:BYTE
□OPEN #PATH,"/D0/SYS/ERRMSG":READ
□WHILE EOF(#PATH)<>TRUE DO
□READ #PATH,A
□PRINT A
□ENDWHILE
□CLOSE #PATH
□END
```

OR Performs a Boolean OR operation

Syntax: ***operand1* OR *operand2***

Function: Performs an OR operation on two or more values, returning a Boolean value of either TRUE or FALSE.

Parameters:

operand1
operand2 Either numeric or string values.

Examples:

```
PRINT A>3 OR B>3

PRINT A$="YES" or B$="YES"
```

Sample Program:

This procedure asks you to type a word or phrase, then converts all lowercase characters to uppercase. It uses OR to test for a character in your word or phrase that is outside of the ASCII values for lowercase letters. If it is, the character does not need converting.

```
PROCEDURE uppercase
□DIM PHRASE,NEWSTRING:STRING[80]; CHARACTER:
STRING[1]; T,X:INTEGER
□NEWSTRING="" \PHRASE=""
□PRINT "Type a phrase in lowercase and I will make
it uppercase."
□INPUT PHRASE
□FOR T=1 TO LEN(PHRASE)
□CHARACTER=MID$(PHRASE,T,1)
□X=ASC(CHARACTER)
□IF X<97 OR X>122 THEN
□NEWSTRING=NEWSTRING+CHARACTER
□ELSE
□X=X-32
□NEWSTRING=NEWSTRING+CHR$(X)
□ENDIF
□NEXT T
□PHRASE=NEWSTRING
□NEWSTRING=""
□PRINT PHRASE
□END
```

PARAM **Establishes variables to receive from another procedure**

Syntax: **PARAM *variable*[,...][:*type*][;*variable*][,...][:*type*] [...]**

Function: Defines the parameters that a *called* procedure expects to receive from the procedure that calls it. When using PARAM, be sure that the total size of each parameter in the calling procedure's RUN statement is the same as the defined size in the called procedure's PARAM statement.

Parameters:

variable	A simple variable, an array structure, or a complex data structure.
type	Byte, Integer, Real, Boolean, String, or user defined.

Notes:

- BASIC09 checks the size of each parameter to prevent accidental access to storage other than that assigned to the parameter. However, BASIC09 does not check that parameters are of the proper type. In most cases you must be sure that types evaluated in RUN statements match the types defined in the PARAM statements.

  However, because BASIC09 does not perform type checking, it is possible to perform useful but normally illegal type conversions of identically-sized data structures. For example, you could pass a string of 80 characters to a procedure expecting a byte array of 80 elements. Each character in the string is assigned a corresponding position in the array.

- You declare simple arrays by using the variable name, without a subscript, in a PARAM statement.

- You can declare several variables of the same type by separating them with commas. To separate variables of different types, follow each type group with a colon, the type name, and then a semicolon.
- If you do not include a maximum length for a string variable enclosed in brackets following the type, like this:

  ```
  DIM name:string[25]
  ```

  BASIC09 uses a default length of 32 characters for strings. You can declare shorter or longer lengths, to the capacity of BASIC09's memory.
- Arrays can have one, two, or three dimensions. The PARAM format for dimensioned arrays is the same as for simple variables except you must follow each array name with a subscript, enclosed in parentheses, to indicate its size. The maximum array size is 32767.

  Arrays can be either of the standard BASIC09 type, or of a user-defined type. To create your own data types for simple variables, arrays, and complex data structures, see TYPE.

Examples:

```
PARAM NUMBER:INTEGER

PARAM NAME:STRING[25];ADDRESS:STRING[30];ZIP:
INTEGER

PARAM NO1,NO2,NO3:REAL;NO4,NO5,NO6:INTEGER;NO7:
BYTE
```

Sample Program:

The first procedure asks you to enter a decimal number. Then, it asks you to choose whether you want to convert the number to binary or hexadecimal. Depending on your choice, the procedure calls (using RUN) either a procedure named Binary or a procedure named Hex. It passes the number you typed to the appropriate procedure for conversion.

```
PROCEDURE convert
□DIM NUMBER,CHOICE:INTEGER
□PRINT USING "S80^"; "Hexadecimal - Binary
Conversion Program"
□PRINT
10□INPUT "Number to convert...",NUMBER
□IF NUMBER=0 THEN
□END
□ENDIF
□INPUT "Choose: (1) Binary or (2) Hex...",CHOICE
□ON CHOICE GOTO 20,30
20□RUN BINARY(NUMBER)
□GOTO 10
30□RUN HEX(NUMBER)
□GOTO 10
□END

PROCEDURE binary
□DIM NUM,X,STORAGE:INTEGER; BI:STRING;
ARRAY(50,2):STRING
□PARAM NUMBER:INTEGER
□COUNT=0
□BI="" \NUM=0 \X=0 \STORAGE=0
□NUM=LOG10(NUMBER)/.3
□NUM=2^NUM \STORAGE=NUMBER
□REPEAT
□X=NUMBER/NUM
□IF X>0 THEN
□BI=BI+"1"
□NUMBER=MOD(NUMBER,NUM)
□ELSE
□BI=BI+"0"
□ENDIF
□NUM=NUM/2
□UNTIL NUM<=1
□IF NUMBER>0 THEN
□BI=BI+"1"
□ELSE
□BI=BI+"0"
□ENDIF
□PRINT STORAGE; " = "; BI; " in binary."
□PRINT
□END
```

```
PROCEDURE hex
□DIM NUM,X,STORAGE:INTEGER; TABLE,HX:STRING;
ARRAY(50,2):STRING
□PARAM NUMBER:INTEGER
□TABLE="123456789ABCDEF"
□HX="" \NUM=0 \X=0 \STORAGE=0
□NUM=LOG10(NUMBER)/1.2
□NUM=16^NUM \STORAGE=NUMBER
□REPEAT
□X=NUMBER/NUM
□IF X>0 THEN
□HX=HX+MID$(TABLE,X,1)
□NUMBER=MOD(NUMBER,NUM)
□ELSE HX=HX+"0"
□ENDIF
□NUM=NUM/16
□UNTIL NUM<=1
□IF NUMBER>0 THEN
□HX=HX+MID$(TABLE,NUMBER,1)
□ELSE
□HX=HX+"0"
□ENDIF
□PRINT STORAGE; " = "; HX; " in hexadecimal."
□PRINT
□END
```

PAUSE Suspends execution and enters Debug

Syntax: **PAUSE *text***

Function: Suspends the execution of a procedure and causes BASIC09 to enter the DEBUG mode. If you include text with the PAUSE command, it is displayed on the screen.

Place PAUSE statements in a program temporarily to observe the way in which the procedure operates and to track down programming errors. When the procedure is operating correctly, remove the PAUSE statement.

After using DEBUG, you can continue execution of the *paused* procedure with the CONT command.

Parameters:

text	A message you want PAUSE to display on the screen when BASIC09 executes the statement.

Examples:

```
PAUSE

PAUSE The array is now full.
```

PEEK Returns the value in a memory location

Syntax: PEEK(***mem***)

Function: Returns the value of a memory byte as a decimal integer. The value returned is in the range 0 to 255. PEEK is the complement of the POKE statement.

See also ADDR.

Parameters:

mem	An integer value representing the location of the memory byte you want to examine. The memory byte is relative to the current process's address space.

Examples:

```
PRINT PEEK(15250)

MEMVAL = PEEK(4450)
```

Sample Program:

This procedure asks you to type a phrase in uppercase characters. It then uses ADDR to locate the area in memory where BASIC09 stores the phrase. Next, it reads each character from memory with PEEK, converts it to lowercase if necessary, and pokes the new value back into the same location. When the procedure displays the contents of the phrase, it is all lowercase.

```
PROCEDURE lowercase
□DIM LOC,T:INTEGER; PHRASE:STRING[80]
□PRINT "Type a phrase in UPPERCASE and I'll make
it lowercase."
□INPUT PHRASE
□LOC=ADDR(PHRASE)
□FOR T=LOC TO LOC+LEN(PHRASE)
□X=PEEK(T)
□IF X>32 AND X<91 THEN
□X=X+32
□POKE T,X
□ENDIF
□NEXT T
□PRINT PHRASE
□END
```

PI Returns the value of pi

Syntax: PI

Function: Returns the constant value 3.14159265.

Parameters: None

Examples:

```
PRINT "The area of a circle with a radius of 6
inches is ";PI*6^2
```

Sample Program:

This procedure uses the formula (PI+2)/15 as a basis for calculating a screen position. Taking the sine of the formula, it prints a sine wave of asterisks down the screen.

```
PROCEDURE picalc
□DIM FORMULA,CALCULATE,POSITION:REAL
□SHELL "DISPLAY 0C"
□FORMULA=(PI+2)/15
□CALCULATE=FORMULA
□SHELL "TMODE -PAUSE"
□FOR T=0 TO 100
□CALCULATE=CALCULATE+FORMULA
□POSITION=INT(SIN(CALCULATE)*10+16)
□PRINT TAB(POSITION); "*"
□NEXT T
□SHELL "TMODE PAUSE"
□END
```

POKE Stores a value in a memory location

Syntax: POKE *mem,value*

Function: Stores a value at the specified memory address, relative to the current process's address space. *Mem* is an absolute address at which BASIC09 stores a byte type value. POKE is the complement of the PEEK statement.

You should use care when using POKE. Because it changes the value in memory, a POKE to the wrong portion of memory could cause OS-9, BASIC09, or your procedures to malfunction until you reboot the system.

See also ADDR.

Parameters:

mem	An integer value representing the location of the memory byte you want to change.
value	The value to store in the specified memory location.

Examples:

```
POKE 15250,13
```

Sample Program:

This procedure asks you to type a phrase in uppercase characters. It then uses ADDR to locate the area in memory where BASIC09 stores the phrase. Next, it reads each character from memory, converts it to lowercase if necessary, and uses POKE to store the new value back in the same location. When the procedure next displays the contents of the phrase, it is all lowercase.

```
PROCEDURE lowercase
□DIM LOC,T:INTEGER; PHRASE:STRING[80]
□PRINT "Type a phrase in UPPERCASE and I'll make
it lowercase."
□INPUT PHRASE
□LOC=ADDR(PHRASE)
□FOR T=LOC TO LOC+LEN(PHRASE)
□X=PEEK(T)
□IF X>32 AND X<91 THEN
□X=X+32
□POKE T,X
□ENDIF
□NEXT T
□PRINT PHRASE
□END
```

POS Returns cursor's column position

Syntax: POS

Function: Returns the current column position of the cursor.

Parameters: None

Examples:

```
PRINT POS
```

Sample Program:

This procedure is a simple typing program that uses POS to make sure that words are not split when you type to the end of the screen. After you type 25 characters on a line, the procedure breaks the line at the next space character.

```
PROCEDURE wordwrap
□DIM CHARACTER:STRING[1]
□PRINT USING "S32^"; "Word Wrap Program"
□PRINT USING "S32^"; "Press [CTRL][C] to Exit"
□PRINT
□SHELL "TMODE -ECHO"
□WHILE CHARACTER<>" " DO
□GET #1,CHARACTER
□PRINT CHARACTER;
□IF POS>25 AND CHARACTER=" " THEN
□PRINT CHR$(13)
□ENDIF
□ENDWHILE
□SHELL "TMODE ECHO"
□END
```

PRINT **Displays text**

Syntax: **PRINT [#*path*] [TAB(*pos*);] *data*[;*data...*]**

Function: Prints numeric or string data on the video display unless another path is specified.

Parameters:

path	The number corresponding to an opened device or file. If you do not specify *path*, the default is #1, the video screen (standard output device). To print to another device or file, first OPEN a path to that file or device (see OPEN).
pos	A column number that tells TAB where to begin printing. Specify any number from 0 to the width of your video display.
data	Any numeric or string constant or variable. Enclose string constants within quotation marks. All data items must be separated by a semicolon or comma.

Notes:

- If you specify more than one data item in the statement, separate them with commas or semicolons.
- If you use commas, PRINT automatically advances to the next tab *zone* before printing the next item. In BASIC09, tab zones are 16 characters apart.
- If you use semicolons or spaces to separate data items, BASIC09 prints the items without any spaces between them. BASIC09 begins the next print item immediately following the end of the last print item.
- If you end a print item without any trailing punctuation, PRINT begins printing at the beginning of the next line.

- If the data being printed is longer than the display screen width, PRINT moves to the next line and continues printing the data.
- TAB causes BASIC09 to begin displaying the specified data at the column position specified by TAB. If the output line is already past the specified TAB position, PRINT ignores TAB.
- You can concatenate items for printing using the plus (+) symbol, for example: `print "hello "+name$+" "+lastname$`.
- PRINT displays REAL numbers with nine or fewer digits in regular format. It displays REAL numbers with more than nine digits in exponential format. For example, 1073741824 is displayed as `1.07374182E+09`.
- You must enclose string constants within quotation marks.

Examples:

```
PRINT A$

PRINT "Menu Items"

PRINT COUNT

PRINT VALUE,TEMP+(n/2.5),LOCATION$

PRINT #PRINTER_PATH,"The result is ";NUMBER

PRINT #OUTPATH FMT$,COUNT,VALUE

PRINT "what is"+NAME$+"'s age? ";

PRINT "INDEX: ";I;TAB(25);"VALUE:  ";VALUE
```

Sample Program:

This procedure asks you to type a word or phrase, then displays it backwards by reading each character from end to beginning and using PRINT to display it on the screen.

```
PROCEDURE reverse
□DIM PHRASE,TITLE:STRING; T,BEGIN:INTEGER
□DIM INSTRUCTIONS:STRING[43]
□TITLE="Word Reversing Program"
□INSTRUCTIONS="Type a word or phrase you want to
reverse: "
□PRINT TITLE
□PRINT "________________________________"
□WHILE PHRASE<>"" DO
□PRINT
□PRINT INSTRUCTIONS
□INPUT PHRASE
□BEGIN=LEN(PHRASE)
□PRINT "This is how your phrase looks backwards:"
□FOR T=BEGIN TO 1 STEP -1
□PRINT MID$(PHRASE,T,1);
□NEXT T
□PRINT
□ENDWHILE
□END
```

PRINT USING **Displays formatted text**

Syntax: **PRINT [#*path*] USING [*format*,] *data*[;*data*...]**

Function: Prints data using a format you specify. This statement is especially useful for printing report headings, accounting reports, checks, or any document requiring a specific format. USING is actually an extension of the PRINT statement; therefore, the same rules that apply to the PRINT statement also apply to the PRINT USING statement (see PRINT).

Parameters:

path	The number corresponding to an opened device or file. If you do not specify *path*, the default is #1, the video screen (standard output device). To print to another device or file, first OPEN a path to that file or device (see OPEN).
format	An expression specifying the arrangement of the displayed data.
data	Any numeric or string constant or variable. Always enclose string constants within quotation marks. Each data item must be separated by semicolons or commas.

Notes:

Each PRINT USING format specifier begins with a single *identifier* letter that specifies the type of format, as shown in the following table:

B	Boolean format
E	exponential format
H	hexadecimal format
I	integer format
R	real format
S	string format

Follow the identifier letter with a constant number that specifies the field width. This number indicates the exact number of print columns the output occupies. It must allow for both the data and any *overhead* characters, such as sign characters, decimal points, exponents, and so on.

Optionally, you can add a justification indicator to the format expression. The indicators are <, >, and ^. The meaning of these indicators varies, depending on the format type in which you use them. See the format type descriptions for specific information.

Note: Do not use any spaces within format expressions.

The following are the format type descriptions:

Real

Use this format for real, integer, or byte type numbers. The total field width specification must include two overhead positions for the sign and decimal point. The field width has two parts, separated by a period. The first part specifies the integer portion of the field. The second part specifies how many fractional digits to display to the right of the decimal point.

If a number has more significant digits than the field allows, BASIC09 uses the undisplayed digits to round the number within the correct field width.

The justification modes are:

- < Left justify with leading sign and trailing spaces. This is the default if you omit a justification indicator.
- > Right justify with leading spaces and sign.
- ^ Right justify with leading spaces and trailing sign (financial format).

Some examples and their results are:

```
PRINT USING "R8.2<",5678.123    5678.12
PRINT USING "R8.2>",5678.123    5678.12
PRINT USING "R8.2>",12.3          12.30
PRINT USING "R8.2>",-555.9      -555.90
PRINT USING "R10.2^",-6722.4599  6722.46-
```

Exponential

Use this format to display real, integer, or byte values in the scientific notation format—using a mantissa and decimal exponent. The field has two parts: the first part must allow for six overhead positions for the mantissa sign, decimal point, and exponent characters.

The justification modes are:

- < Left justify with leading sign and trailing spaces. This is the default if you omit a justification indicator.
- > Right justify with leading spaces and sign.

Some examples and their results are:

```
PRINT USING "E12.3",1234.567     1.235E+03
PRINT USING "E13.6>",-.001234  -1.234000E-03
PRINT USING "E18.5>",123456789        1.23457E+08
```

Integer

Use this format to display integer, byte, or real type numbers in an integer or byte format. The field width must allow for one position of overhead for the sign.

The justification modes are:

- < Left justify with leading sign and trailing spaces. This is the default if you omit a justification indicator.
- > Right justify with leading spaces and sign.
- ^ Right justify with leading sign and zeroes.

Some examples and their results are:

```
PRINT USING "I4<",10    10
PRINT USING "I4<",10      10
PRINT USING "I4^",-10  -010
```

Hexadecimal

Use this format to display any data type in hexadecimal notation. The field width specification determines the number of hexadecimal characters BASIC09 displays. If the data to display is string type, this function displays the ASCII value of each character in hexadecimal.

The justification modes are:

- < Left justify with trailing spaces. This is the default if you omit a justification indicator.
- > Right justify with leading spaces.
- ^ Center digits.

The number of bytes of memory used to represent data varies according to data type. The following chart suggests field widths for specific data types:

Type	**Memory Bytes**	**Field Width To Specify**
Boolean and Byte	1	2
Integer	2	4
Real	5	10
String	1 per character	2 times the string length

Some examples and their results are:

```
PRINT USING "H4",100        0064
PRINT USING "H4",-1         FFFF
PRINT USING "H8^","ABC"     414243
```

String

Use this format to display string data of any length. The field width specifies the total field size. If the string to display is shorter than the field size, PRINT USING pads it with spaces according to the justification mode. If the string to display is longer than the specified field width, PRINT USING truncates the right portion of the string.

The justification modes are:

- < Left justify with trailing spaces. This is the default if you omit a justification indicator.
- > Right justify with leading spaces.
- ^ Center characters.

Some examples and their results are:

```
PRINT USING "S9<","HELLO"   HELLO
PRINT USING "S9>","HELLO"       HELLO
PRINT USING "S9^","HELLO"     HELLO
```

Boolean

Use this format to display Boolean expression results. BASIC09 converts the result of the expression to the strings "True" or "False." The format and results are identical to STRING formats. The justification modes are:

- `<` Left justify with trailing spaces. This is the default if you omit a justification indicator.
- `>` Right justify with leading spaces.
- `^` Center characters.

If A=5 and B=6, some examples and their results are:

```
PRINT USING "B9<",A<B   True
PRINT USING "B9>",A>B       False
PRINT USING "B9^",A=B     False
```

Control Specifiers

You can also use *control specifiers* within PRINT USING formats. The three specifiers are:

T*n* Tab. *n* specifies a tab column at which to display the next data.

X*n* Spaces. *n* specifies a number of spaces to insert.

'*text*' Constant string. *text* is a string that is constant to the format.

An example and its result is:

```
PRINT USING "'Address',X1,H4,X4,'Data',X1,H2",
1000,100

Address 03E8    Data 64
```

Repeat

You can repeat identical sequences of specifications using parentheses within a format specification. Enclose the group of specifications you wish to repeat, preceded by a repetition count, such as:

`"2(X2,r10.5)"` in place of `"X2,R10.5,X2,R10.5"`

`"2(I2,2(X1,S4))"` in place of `"I2,X1,S4,X1,S4,I2,X1, S4,X1,S4"`

Sample Program:

This program looks at memory locations 32000 to 32010 and displays their contents in decimal, hexadecimal, and binary. PRINT USING formats the display in columns.

```
PROCEDURE memlook
□DIM NUMBER,T,MEM,VALUE:INTEGER
□DIM X,NUM:INTEGER; CHARACTER,BI:STRING
□PRINT "□Addr.□Dec.□Hex.□Bin□□□□□□□ASCII"
□FOR Z=32000 TO 32010
□BI=""
□NUMBER=PEEK(Z)
□IF NUMBER>0 THEN
□GOSUB 100
□ENDIF
□IF PEEK(Z)<32 THEN
□CHARACTER=""
□ELSE
□CHARACTER=CHR$(PEEK(Z))
□ENDIF
□IF PEEK(Z)>0 THEN
□PRINT USING "I6<,T7,I4<,X2,H4<,X1,S8<,X2,S1",Z,
PEEK(Z),PEEK(Z),BI,CHARACTER
□ELSE PRINT USING "I6<,T7,I4<,X2,H4<,X1,S8>,X2,
S1",Z,0,0,"0000"," "
□ENDIF
□NEXT Z
□END
```

```
100□NUM=LOG10(NUMBER)/.3
□NUM=2^NUM
□REPEAT
□X=NUMBER/NUM
□IF X>0 THEN BI=BI+"1"
□NUMBER=MOD(NUMBER,NUM)
□ELSE BI=BI+"0"
□ENDIF
□NUM=NUM/2
□UNTIL NUM<=1
□IF NUMBER>0 THEN
□BI=BI+"1"
□ELSE BI=BI+"0"
□ENDIF
□RETURN
□END
```

PUT Writes to a direct access file

Syntax: **PUT *#path,data***

Function: Writes a fixed-size binary data record to a file or device. Use PUT to store data in random access files.

Although you usually use PUT with files, you can also use it to send data to a device.

For information about storing data in random access files, see Chapter 8, "Disk Files". Also, see GET, SEEK, and SIZE.

Parameters:

path	A variable name you chose to use in an OPEN or CREATE statement that stores the number of the path to the file or device to which you are directing data.
data	Either a variable containing the data you want to send or a string of data.

Examples:

```
PUT #PATH,DATA$

PUT INPUT,ARRAY$
```

Sample Program:

This procedure is a simple inventory data base. You type in the information for an item name, list cost, actual cost, and quantity. Using PUT, the procedure stores data in a file named Inventory.

```
PROCEDURE inventory
□TYPE INV_ITEM=NAME:STRING[25]; LIST,COST:REAL;
QTY:INTEGER
□DIM INV_ARRAY(100):INV_ITEM
□DIM WORK_REC:INV_ITEM
□DIM PATH:BYTE
□ON ERROR GOTO 10
```

```
□DELETE "inventory"
10□ON ERROR
□CREATE #PATH,"inventory"
□WORK_REC.NAME=""
□WORK_REC.LIST=0
□WORK_REC.COST=0
□WORK_REC.QTY=0
□FOR N=1 TO 100
□PUT #PATH,WORK_REC
□NEXT N
□LOOP
□INPUT "Record number? ",recnum
□IF recnum<1 OR recnum>100 THEN
□PRINT
□PRINT "End of Session"
□PRINT
□CLOSE #PATH
□END
□ENDIF
□INPUT "Item name? ",WORK_REC.NAME
□INPUT "List price? ",WORK_REC.LIST
□INPUT "Cost price? ",WORK_REC.COST
□INPUT "Quantity? ",WORK_REC.QTY
□SEEK #PATH,(recnum-1)*SIZE(WORK_REC)
□PUT #PATH,WORK_REC
□ENDLOOP
□END
```

RAD Returns trigonometric calculations in radians

Syntax: **RAD**

Function: Set a procedure's *state flag* so that a procedure uses radians in SIN, COS, TAN, ACS, ASN, and ATN functions. Because this is the BASIC09 default, you do not need to use the RAD statement unless you previously used a DEG statement in the procedure.

Parameters: None

Examples:

```
RAD
```

Sample Program:

This program calculates sine, cosine, and tangent for a value you supply. It calculates one set of results in degrees, using DEG, and the second set of results in radians using RAD.

```
PROCEDURE trigcalc
□DIM ANGLE:REAL
□DEG
□INPUT "Enter the angle of two sides of a
triangle...",ANGLE
□PRINT
□PRINT "□□□□□□□□□□Angle","SINE","COSINE","TAN"
□PRINT "□□□□□□□□□□------------------------------------
------------------"
□PRINT "Degrees = "; ANGLE,SIN(ANGLE),COS(ANGLE),
TAN(ANGLE)
□RAD
□PRINT "Radians = "; ANGLE,SIN(ANGLE),COS(ANGLE),
TAN(ANGLE)
□PRINT
□END
```

READ Reads data from a device or DATA statement

Syntax: **READ [#*path*,] *varname***

Function: Reads either an ASCII record from a sequential file or device, or an item from a DATA statement.

Parameters:

path	A variable containing the path number of the file you want to access. You can also specify one of the standard I/O paths (0, 1, or 2).
varname	The variable in which you want to store the data read from a file, device, or DATA line.

Notes:

The following information deals with reading sequential disk files:

- To read file records, you must first dimension a variable to contain the path number of the file, then use OPEN or CREATE to open a file in the READ or UPDATE access mode. The command begins reading records at the first record in the file. After it reads each item, it updates the pointer to the next item.
- Records can be of any length within a file. Make sure the variable you use to store the records is dimensioned large enough to store each item. If the variable storage is too small, BASIC09 truncates the record to the maximum size for which you dimensioned the variable. If you do not indicate a variable size with the DIM statement, the default is 32 characters.
- BASIC09 separates individual data items in the input record with ASCII null characters. You can also separate numeric items with comma or space character delimiters. Each input record is terminated by a carriage return character.

The following information deals with reading DATA items:

- READ accesses DATA line items sequentially. Each string type item in a DATA line must be surrounded by quotation marks. Items in a DATA line must be separated with commas.
- Each READ command copies an item into the specified variable storage and updates the data pointer to the next item, if any.
- You can independently move the pointer to a selected DATA statement. To do this, use line numbers with the DATA lines See the DATA and RESTORE commands for more information on using this function of READ.

Examples:

```
READ #PATH,DATA

READ #1,RESPONSE$

READ #INPUT,INDEX(X)

FOR T=1 TO 10
READ NAME$(T)
NEXT T
DATA "JIM","JOE","SUE","TINA","WENDY"
DATA "SALL","MICKIE","FRED","MARV","WINNIE"
```

Sample Program:

This procedure puts random values between 1 and 10 into a disk file, then READS the values and uses asterisks to indicate how many times RND selected each value.

```
PROCEDURE randlist
□DIM SHOW,BUCKET:STRING
□DIM T,PATH,SELECT(10),R:INTEGER
□BUCKET="*************************"
□FOR T=1 TO 10
□SELECT(T)=0
□NEXT T
□ON ERROR GOTO 10
□SHELL "DEL RANDFILE"
10□ON ERROR
□CREATE #PATH,"randfile":UPDATE
□FOR T=1 TO 100
□R=RND(9)+1
□WRITE #PATH,R
□NEXT T
□PRINT "Random Distribution"
□SEEK #PATH,0
□FOR T=1 TO 100
□READ #PATH,NUM
□SELECT(NUM)=SELECT(NUM)+1
□NEXT T
□FOR T=1 TO 10
□SHOW=RIGHT$(BUCKET,SELECT(T))
□PRINT USING "S6<,I3<,S2<,S20<","Number",
T,":",SHOW
□NEXT T
□CLOSE #PATH
□END
```

REM Inserts remarks in a procedure

Syntax: **REM [*text*]**
(* [*text*][*)]

Function: Inserts remarks inside a procedure. BASIC09 ignores these remarks; they serve only to document a procedure and its functions. Use remarks to title a procedure, show its creation date, show the name of the programmer, or to explain particular features and operations of a procedure.

Parameters:

text Comments you want to include within a procedure

Notes:

- You can insert remarks at any point in a procedure.
- The second form of REM, using parentheses and asterisks, is compatible with Pascal programming structure.
- When editing programs, you can use the exclamation character "!" in place of the keyword REM.
- BASIC09's initial compilation retains remarks, but the PACK compile command strips them from procedures.

Examples:

```
REM this is a comment

(* Insert text between parentheses and
asterisks*)

(* or use only one parenthesis and asterisk
```

Sample Program:

This procedure uses the various forms of REM to explain its operations.

```
PROCEDURE copydir
□REM Use this program with the
□(* GET sample program to *)
□(* create a file of directory*)
□(* filenames, then copy the*)
□(* files to another directory*)
□DIM PATH,T,COUNT:INTEGER; FILE,JOB,DIRNAME:STRING
□OPEN #PATH,"dirfile":READ (* open the file
□INPUT "Name of new directory...",DIRNAME (* get the directory
□SHELL "MAKDIR "+DIRNAME (* create a newdirectory
□SHELL "LOAD COPY"
□WHILE NOT(EOF(#PATH)) DO
□READ #PATH,FILE (* get a filename
□JOB=FILE+" "+DIRNAME+"/"+FILE (* create the COPY syntax
□ON ERROR GOTO 10
□PRINT "COPY "; JOB (* display the operation
□SHELL "COPY "+JOB (* copy the file
10□ON ERROR
□ENDWHILE
□CLOSE #PATH
□END
```

REPEAT/UNTIL

Establishes a loop/Terminates on specified condition

Syntax: **REPEAT**
procedure lines
UNTIL *expression*

Function: Establishes a loop that executes the encompassed procedure lines until the result of the expression following UNTIL is true. Because the loop is tested at the bottom, the lines within the loop are executed at least once.

Parameters:

expression	A Boolean expression (returns either True or False).
procedure lines	Statements you want to repeat until *expression* returns False.

Examples:

```
REPEAT
COUNT = COUNT+1
UNTIL COUNT > 100
```

```
INPUT X,Y
REPEAT
X = X-1
Y = Y-1
UNTIL X<1 OR Y<0
```

Sample Program:

The procedure sorts a disk file. In this case, it is written to sort the diskfile created by the GET sample program—a directory listing. It uses a REPEAT/UNTIL loop to compare a string in the file with the first string in the file. If the first string is greater than the comparison string, the procedure swaps them.

```
PROCEDURE dirsort
□DIM BTEMP:BOOLEAN; TEMP,FILES(125):STRING; TOP,
BOTTOM,M,N:INTEGER
□DIM T,X,PATH:INTEGER
□FOR T=1 TO 125
□FILES(T)=""
□NEXT T
□T=0
□OPEN #PATH,"dirfile":READ
□PRINT "LOADING:"
□WHILE NOT(EOF(#PATH)) DO
□T=T+1
□READ #PATH,FILES(T)
□ENDWHILE
□TOP=T
□BOTTOM=1
□PRINT "SORTING: ";
10□N=BOTTOM
□M=TOP
□PRINT ".";
□LOOP
□REPEAT
□BTEMP=FILES(N)<FILES(TOP)
□N=N+1
□UNTIL NOT(BTEMP)
□N=N-1
□EXITIF N=M THEN
□ENDEXIT

□TEMP=FILES(M)
□FILES(M)=FILES(N)
□FILES(N)=TEMP
□N=N+1
□EXITIF N=M THEN
□ENDEXIT
```

```
 ENDLOOP
 IF N<>TOP THEN
 IF FILES(N)<>FILES(TOP) THEN
 TEMP=FILES(N)
 FILES(N)=FILES(TOP)
 FILES(TOP)=TEMP
 ENDIF
 ENDIF

 IF BOTTOM<N-1 THEN
 TOP=N-1
 GOTO 10
 ENDIF
 IF N+1<TOP THEN
 BOTTOM=N+1
 GOTO 10
 ENDIF
 CLOSE #PATH
 DELETE "dirfile"
 CREATE #PATH,"dirfile":WRITE
 PRINT
 FOR Z=1 TO T
 WRITE #PATH,FILES(Z)
 PRINT FILES(Z),
 NEXT Z

 CLOSE #PATH
 END
```

RESTORE **Resets READ pointer**

Syntax: **RESTORE *linenumber***

Function: Sets the pointer for the READ command to the specified line number. RESTORE without a line number sets the data pointer to the first data statement in the procedure.

READ assigns the items in a DATA statement to variable storage. When you read an item, the pointer automatically advances to the next item. Using RESTORE you can skip backward or forward to data items at a specific line number.

Parameters:

linenumber The line number of the DATA items you want READ to access next.

Examples:

```
RESTORE 100
```

Sample Program:

This procedure draws a box on the screen. It uses RESTORE to repeat the data in line 20 to create the sides of the box.

```
PROCEDURE box
□DIM LINE:STRING
□READ LINE
□PRINT LINE
□FOR T=1 TO 10
□RESTORE 20
□READ LINE
□PRINT LINE
□NEXT T
□RESTORE 10
□READ LINE
□PRINT LINE
10□DATA "------------------------"
20□DATA "□□□□□□□□□□□□□□□□□□□□□□□"
□END
```

RETURN **Returns from subroutine**

Syntax: **RETURN**

Function: Returns procedure execution to the line immediately following the last GOSUB statement.

Every subroutine you access with GOSUB must contain a RETURN statement. You can call a subroutine in this manner as many times as you want.

Parameters: None

Sample Program:

This procedure draws a design of asterisks down the display screen. It uses MOD to send execution to a series of PRINT USING routines over and over. Each PRINT USING routine sends execution back to the main routine with a RETURN statement.

```
PROCEDURE stars
□DIM T:INTEGER
□SHELL "TMODE -PAUSE"
□FOR T=1 TO 100
□ON MOD(T,8)+1 GOSUB 10,20,30,40,50,60,70,80
□NEXT T
□SHELL "TMODE PAUSE"
□END
10□PRINT USING "S10^","*" \ RETURN
20□PRINT USING "S10^","**" \ RETURN
30□PRINT USING "S10^","***" \ RETURN
40□PRINT USING "S10^","****" \ RETURN
50□PRINT USING "S10^","*****" \ RETURN
60□PRINT USING "S10^","****" \ RETURN
70□PRINT USING "S10^","***" \ RETURN
80□PRINT USING "S10^","**" \ RETURN
□END
```

RIGHT$ Returns specified rightmost portion of a string

Syntax: **RIGHT$(*string,length*)**

Function: Returns the specified number of characters from the right portion of the specified string. If *length* is the same as or greater than the number of characters in *string*, then RIGHT$ returns all of the characters in the string.

Parameters:

string	A sequence of string type characters or a variable containing a sequence of string type characters.
length	The number of characters you want to access.

Examples:

```
PRINT RIGHT$("HOTDOG",3)

PRINT RIGHT$(A$,6)
```

Sample Program:

```
PROCEDURE lastname
□DIM NAMES:STRING; LASTNAME:STRING[10]
□PRINT "Here are the last names:"
□FOR T=1 TO 10
□READ NAMES
□POINTER=SUBSTR(" ",NAMES)
□POINTER=LEN(NAMES)-POINTER
□LASTNAME=RIGHT$(NAMES,POINTER)
□PRINT LASTNAME
□NEXT T
□DATA "Joe Blonski","Mike Marvel","Hal Skeemish",
"Fred Langly"
□DATA "Jane Misty","Wendy Paston","Martha
Upshong","Jacqueline Rivers"
□DATA "Susy Reetmore","Wilson Creding"
□END
```

RND Returns a random value

Syntax: **RND(*number*)**

Function: Returns a random real value in the following ranges:

If *number* = 0 then range = 0 to 1
If *number* > 0 then range = 0 to *number*

The values produced by RND are not truly random numbers, but occur in a predictable sequence. Specifying a number less than 0 begins the sequence over.

Parameters:

number A numeric constant, variable, or expression.

Examples:

```
PRINT RND(5)

PRINT RND(A)

PRINT RND(A*5)
```

Sample Program:

This procedure presents addition problems for you to solve. It uses RND to select two numbers between 0 and 20.

```
PROCEDURE addition
□DIM A,B,ANSWER,C:INTEGER
□FOR T=1 TO 5
□A=RND(2Ø)
□B=RND(2Ø)
□C=A+B
□PRINT USING "'What is:□',I3>",A
□PRINT USING "'□□□□□□□+□',I3>",B
□PRINT "□□□□□□□□□------"
□INPUT "□□□□□□□□□□",ANSWER
□IF ANSWER=C THEN
□PRINT "CORRECT"
□ELSE
□PRINT "WRONG"
□ENDIF
□PRINT
□NEXT T
□END
```

RUN Executes another procedure

Syntax: RUN ***procname*** [(***param***[,***param***,...])]

Function: Calls a procedure for execution, passing the specified parameters to the called procedure. When the called procedure ends, execution returns to the calling procedure, beginning at the statement following the RUN statement.

RUN can call a procedure existing within the workspace, a procedure previously compiled by the PACK command, or a machine language procedure outside the workspace.

Parameters:

procname — The name of the procedure to execute. The *procname* can be the literal name of the procedure to execute, or it can be a variable name containing the procedure name.

param — One or more parameters that the called program needs for execution. The parameters can be variables or constants, or the names of entire arrays or data structures.

Notes:

- You can pass all types of data to a called program except byte type. However, you can pass byte arrays.
- If a parameter is a constant or expression, BASIC09 passes it *by value*. That is, BASIC09 evaluates the constant or expression and places it in temporary storage. It passes the address of the temporary storage location to the called procedure. The called program can change the passed values, but the changes are not reflected in the calling procedure.
- If a parameter is the name of a variable, array, or data structure, BASIC09 passes it to the called program by *reference*. That is, it passes the address of the variable storage to the called procedure. Thus, the value can be changed by the receiving procedure, and these changes are reflected in the calling procedure.

- If the procedure named by RUN is not in the workspace, BASIC09 looks outside the workspace. If it cannot find it there, it looks in the current execution directory for a disk file with the proper name. If the file is on disk, BASIC09 loads and executes it, regardless of whether it is a packed BASIC09 program or a machine language program.

  If the program is a machine language module, BASIC09 executes a JSR (jump to subroutine) instruction to its entry point and executes it as 6809 native code. The machine language program returns to the original calling procedure by executing a RTS (return from subroutine) instruction.

- After you call an external procedure, and no longer need it, use KILL to remove it from memory to free space for other operations.

- Machine language modules return error status by setting the C bit of the MPU condition codes register, and by setting the B register to the appropriate error code.

Examples:

```
RUN CALCULATE(10,20,ADD)

RUN PRINT(TEXT$)
```

Sample Program:

Makelist creates and displays a list of fruit. Next, it asks you to type a word to insert. After you type and enter a new word, Makelist uses RUN to call a second procedure named Insert to look through the list and insert the new word in alphabetical order. After each insertion, the procedure asks for another word. Press only ENTER to terminate the program.

```
PROCEDURE makelist
□DIM LIST(25),NEWORD,TEMPWORD:STRING[15]
□DIM T,LAST:INTEGER
□LAST=10
□PRINT "This is your list..."
□FOR T=1 TO LAST
□READ LIST(T)
□PRINT LIST(T),
□NEXT T
□LOOP
```

```
□PRINT
□PRINT
□INPUT "Type a word to insert...",NEWORD
□EXITIF NEWORD="" THEN
□PRINT
□END "I've ended the session at your request..."
□ENDEXIT
□RUN Insert(LIST,NEWORD,LAST)
□PRINT
□PRINT "This is your new list..."
□FOR T=1 TO LAST
□PRINT LIST(T),
□NEXT T
□PRINT
□ENDLOOP
□DATA "APPLES","BANANAS","CANTALOUPE"
□DATA "DATES","GRAPES","LEMONS"
□DATA "MANGOS","PEACHES","PLUMS"
□DATA "PEARS"

PROCEDURE insert
□PARAM LIST(25),NEWORD:STRING[15]
□PARAM LAST:INTEGER
□DIM TEMPWORD:STRING[15]
□DIM T,X:INTEGER
□T=1
□WHILE NEWORD>LIST(T) DO
□T=T+1
□ENDWHILE
□FOR X=T TO LAST
□TEMPWORD=LIST(X)
□LIST(X)=NEWORD
□NEWORD=TEMPWORD
□NEXT X
□LAST=LAST+1
□LIST(LAST)=NEWORD
□END
```

SEEK Resets the direct-access file pointer

Syntax: **SEEK *#path,number***

Function: Changes the file pointer address in a disk file. The pointer indicates the location in a file for the next READ or WRITE operation.

You usually use SEEK with random access files to move the pointer from one record to another, in any order. You can also use SEEK with sequential access files to *rewind* the pointer to the beginning of the file (to the first item or record).

For information about storing data in random access files, see Chapter 8, "Disk Files." Also see PUT, GET, and SIZE.

Parameters:

path A variable name you choose in which BASIC09 stores the number of the path it opens to the file you specify.

number The item or record number you want to access. If you are rewinding a sequential access file, specify a *number* of 0.

Examples:

```
SEEK #PATH,0

SEEK #OUTFILE,A

SEEK #INDEX,LOCATION*SIZE(INVENTORY)
```

Sample Program:

This procedure creates a file named Test1, then writes 10 lines of data into it. Next, it reads the lines from the file and displays them. It uses SEEK to both store and extract the lines in blocks of 25 characters.

```
PROCEDURE makelines
□DIM LENGTH:BYTE
□DIM LINE:STRING[25]
□DIM PATH:BYTE
□LENGTH=25
□BASE Ø
□ON ERROR GOTO 1Ø
□DELETE "test1"
1Ø□ON ERROR

□CREATE #PATH,"test1":WRITE

□FOR T=Ø TO 9
□READ LINE$
□SEEK #PATH,LENGTH*T
□PUT #PATH,LINE$
□NEXT T
□CLOSE #PATH

□OPEN #PATH,"test1":READ
□FOR T=9 TO Ø STEP -1
□SEEK #PATH,LENGTH*T
□GET #PATH,LINE
□PRINT LINE
□NEXT T
□CLOSE #PATH
□END

□DATA "This is test line #1"
□DATA "This is test line #2"
□DATA "This is test line #3"
□DATA "This is test line #4"
□DATA "This is test line #5"
□DATA "This is test line #6"
□DATA "This is test line #7"
□DATA "This is test line #8"
□DATA "This is test line #9"
□DATA "This is test line #1Ø"
```

SGN Returns a value's sign

Syntax: SGN(***number***)

Function: Determines whether a number's sign is positive or negative.

If *number* is less than 0, then SGN returns -1. If *number* equals 0, then SGN returns 0. If *number* if greater than 0, then SGN returns 1.

Parameters:

number The value for which you want to determine the sign.

Examples:

```
PRINT SGN(-22)

PRINT SGN(A)

PRINT SGN(44-A)
```

Sample Program:

This procedure uses SGN to create half sine waves down the screen. SGN tests when the SIN calculation results are positive.

```
PROCEDURE halfsine
□DIM FORMULA,CALCULATE,POSITION:REAL
□SHELL "DISPLAY 0C"
□FORMULA=(PI+2)/15
□CALCULATE=FORMULA
□SHELL "TMODE -PAUSE"
□FOR T=0 TO 100
□CALCULATE=CALCULATE+FORMULA
□POSITION=INT(SIN(CALCULATE)*10+16)
□IF SGN(SIN(CALCULATE))>0 THEN
□PRINT TAB(POSITION); "*"
□ENDIF
□NEXT T
□SHELL "TMODE PAUSE"
□END
```

SHELL Forks another shell

Syntax: SHELL ["*string*"][+"*string*"...][+ *variable*]
[+ *variable...*]

Function: Executes OS-9 commands or programs from within a BASIC09 procedure. Using SHELL, you can access OS-9 functions, including multiprogramming, utilities, commands, terminal and input/output control, and so on.

When you use the SHELL command, OS-9 creates a new process to handle the commands you provide. If you specify an operation, BASIC09 evaluates the expression and passes it to the shell for execution. If you do not specify an operation, BASIC09 temporarily halts, and the shell process displays prompts and accepts commands in the normal manner. In this case, press CTRL BREAK to return to BASIC09.

When the shell process terminates, BASIC09 becomes active and resumes execution at the statement following the SHELL statement.

Parameters:

string Any OS-9 command or function. String constants must be enclosed in quotation marks. Concatenate string constants and string variables using a plus symbol (+).

variable Any string variable containing an OS-9 command or function.

Examples:

```
SHELL "COPY FILE1 FILE2"

SHELL "COPY FILE1 FILE2&"

SHELL "COPY "+FILE$+" "+DIRNAME+"/"FILE$

SHELL "LIST DOCUMENT"

SHELL "KILL "+STR$(N)
```

Sample Program:

You must use this procedure with the GET sample program. Using the two programs together enables you to copy all the files from one directory to another directory. The GET sample program reads the files in a directory and stores them in a file named Dirfile. This procedure reads the filenames from Dirfile and uses SHELL to copy them to the directory you specify.

```
PROCEDURE copyutil
□DIM PATH,T,COUNT:INTEGER; FILE,JOB,DIRNAME:STRING
□OPEN #PATH,"dirfile":READ
□INPUT "Name of new directory...",DIRNAME
□SHELL "MAKDIR "+DIRNAME
□SHELL "LOAD COPY"
□WHILE NOT(EOF(#PATH)) DO
□READ #PATH,FILE
□JOB=FILE+" "+DIRNAME+"/"+FILE
□ON ERROR GOTO 10
□PRINT "COPY "; JOB
□SHELL "COPY "+JOB
10□ON ERROR
□ENDWHILE
□CLOSE #PATH
□END
```

SIN Returns the sine of a number

Syntax: SIN(***number***)

Function: Calculates the trigonometric sine of *number*. You can use the DEG or RAD commands to cause *number* to represent a value in either degrees or radians. Unless you specify DEG, the default is radians. SIN returns a real number.

Parameters:

number	The angle of two sides of a triangle for which you want to find the ratio.

Examples:

```
PRINT SIN(45)
```

Sample Program:

This procedure calculates sine, cosine, and tangent values for a number you type.

```
PROCEDURE ratiocalc
□DEG
□DIM ANGLE:REAL
□INPUT "Enter the angle of two sides of a
triangle...",ANGLE
□PRINT
□PRINT "Angle","SINE","COSINE","TAN"
□PRINT "--------------------------------------
--------------------"
□PRINT ANGLE,SIN(ANGLE),COS(ANGLE),TAN(ANGLE)
□PRINT
□END
```

SIZE Returns the size of a data structure

Syntax: **SIZE(*variable*)**

Function: Returns the size in bytes of a variable, array, or data structure. SIZE is especially useful with random access files to determine the size of records to store in a file. You can also use SIZE to determine the size of variable storage for other purposes.

SIZE returns the size of assigned storage, not necessarily the size of a string. For example, if you dimension a variable for 15 characters, and assign a 10-character string to it, SIZE returns 15, not 10. SIZE returns the total size of arrays. That is, it returns the number of elements multiplied by the size of the elements.

Parameters:

variable The variable, array, or data structure for which you want to find the size.

Examples:

```
RECORDLENGTH = SIZE(A$)
```

```
PRINT "YOUR NAME IS STORED IN A "; SIZE(NAME$);
" CHARACTER STRING."
```

Sample Program:

This procedure creates a simple inventory, stored in a file named Inventory. It uses SIZE to calculate the size of each element to be stored in the file, and to move the pointer to the beginning of each record's storage space.

```
PROCEDURE inventory
□TYPE INV_ITEM=NAME:STRING[25]; LIST,COST:REAL;
QTY:INTEGER
□DIM INV_ARRAY(100):INV_ITEM
□DIM WORK_REC:INV_ITEM
□DIM PATH:BYTE
□ON ERROR GOTO 10
□DELETE "inventory"
10□ON ERROR
□CREATE #PATH,"inventory"
□WORK_REC.NAME=""
□WORK_REC.LIST=0
□WORK_REC.COST=0
□WORK_REC.QTY=0
□FOR N=1 TO 100
□PUT #PATH,WORK_REC
□NEXT N
□LOOP
□INPUT "Record number? ",recnum
□IF recnum<1 OR recnum>100 THEN
□PRINT
□PRINT "End of Session"
□PRINT
□CLOSE #PATH
□END
□ENDIF
□INPUT "Item name? ",WORK_REC.NAME
□INPUT "List price? ",WORK_REC.LIST
□INPUT "Cost price? ",WORK_REC.COST
□INPUT "Quantity? ",WORK_REC.QTY
□SEEK #PATH,(recnum-1)*SIZE(WORK_REC)
□PUT #PATH,WORK_REC
□ENDLOOP
□END
```

SQ Returns the value of a number raised to the power of 2

Syntax: **SQ(*number*)**

Function: Calculates the value of a number raised to the power of 2.

Parameters:

number The number you want raised to the power of 2.

Examples:

```
PRINT SQ(188)

PRINT PI*SQ(R)
```

Sample Program:

This procedure uses SQ in a formula that positions asterisks on the screen in a sine wave pattern.

```
PROCEDURE sinedown
□DIM FORMULA,CALCULATE,POSITION:REAL
□SHELL "DISPLAY ØC"
□FORMULA=(PI+2)/15
□CALCULATE=FORMULA
□SHELL "TMODE -PAUSE"
□FOR T=Ø TO 2ØØ
□CALCULATE=CALCULATE+SQ(FORMULA)
□POSITION=INT(SIN(CALCULATE)*12+16)
□PRINT TAB(POSITION); "*"
□NEXT T
□SHELL "TMODE PAUSE"
□END
```

SQR/SQRT Returns the square root of a number

Syntax: **SQR(*number*)**
SQRT(*number*)

Function: Calculates the square root of a number. SQR and SQRT serve the same function.

Parameters:

number The number for which you want the square root.

Examples:

```
PRINT SQR(188)

PRINT PI*SQRT(R)
```

Sample Program:

This procedure uses SQRT in a formula to position asterisks on the screen in a sine wave pattern.

```
PROCEDURE sqrdown
□DIM FORMULA,CALCULATE,POSITION:REAL
□SHELL "DISPLAY 0C"
□FORMULA=PI/15
□CALCULATE=FORMULA
□SHELL "TMODE -PAUSE"
□FOR T=0 TO 200
□CALCULATE=CALCULATE+SQRT(FORMULA)
□POSITION=INT(SIN(CALCULATE)*12+16)
□PRINT TAB(POSITION); "*"
□NEXT T
□SHELL "TMODE PAUSE"
□END
```

STEP **Establishes the size of increments in a FOR loop**

Syntax:
FOR *variable = init val* TO *end val* [STEP *value*]
[*procedure statements*]
NEXT *variable*

Function: STEP provides an increment value in a FOR/NEXT loop. If you do not specify a STEP value, the loop steps in increments of 1.

BASIC09 executes the lines following the FOR statement until it encounters a NEXT statement. Then it either increases or decreases the initial value by 1 (the default) or by the value given by STEP. If you give the loop an initial value that is greater than the final value, and specify a negative value for STEP, the loop decreases from the initial value to the end value.

Parameters:

variable	Any legal numeric variable name.
init val	Any numeric constant or variable.
end val	Any numeric constant or variable.
value	Any numeric constant or variable.
procedure statements	Procedure lines you want to be executed within the loop.

Examples:

```
FOR COUNTER = 1 to 100 STEP .5
PRINT COUNTER
NEXT COUNTER

FOR X = 10 TO 1 STEP -1
PRINT X
NEXT X

FOR TEST = A TO B STEP RATE
PRINT TEST
NEXT TEST
```

Sample Program:

This procedure reverses the order of characters in a word or phrase you type. It uses STEP to decrement a counter that points to each character in the string in reverse order.

```
PROCEDURE reverse
□DIM PHRASE:STRING; T,BEGIN:INTEGER
□PRINT "Type a word or phrase you want to
reverse:";
□PRINT
□INPUT PHRASE
□BEGIN=LEN(PHRASE)
□PRINT "This is how your phrase looks backwards:"
□FOR T=BEGIN TO 1 STEP -1
□PRINT MID$(PHRASE,T,1);
□NEXT T
□PRINT
□END
```

STOP Terminates a procedure

Syntax: **STOP [“*string*”]**

Function: Causes a procedure to cease execution, print the message “STOP Encountered”, and return control to BASIC09’s command mode. You can also specify additional text to display when BASIC09 encounters STOP.

Use stop when you want a procedure to terminate without entering the DEBUG mode.

Parameters:

string	Text to display when STOP executes.

Examples:

```
STOP "Program terminated before completion."

IF RESPONSE = "Y" THEN STOP "Program terminated
at your request."
ENDIF
```

STR$ Converts numeric data to string data

Syntax: **STR$(*number*)**

Function: Converts a numeric type to a string type. This lets you display the number as a string or use string operators on a number. The conversion replaces the numeric values with the ASCII characters of the number. STR$ is the inverse of the VAL function.

Parameters:

number Any numeric-type data.

Examples:

```
PRINT STR$(1010)

DIM I:INTEGER
I=44
PRINT STR$(I)

DIM B:BYTE
B=001
PRINT STR$(B)

DIM R:REAL
R=1234.56
PRINT STR$(R)
```

Sample Program:

This procedure calculates an exponential value, then adds the necessary number of zeroes to convert it to standard notation. It uses STR$ to convert the number you type to a string type value so that it can use string functions to add the zeroes.

```
PROCEDURE bignum
□DIM C,PLACES,B,SIGN:STRING; EX,COUNT,NEWCOUNT,
DECIMAL:INTEGER
□DIM NEW,ZERO,NEWEST:STRING[100]
□COUNT=-1
□ZERO="0000000000000000000000000000000000000000"
□NEW="" \NEWEST=""
□INPUT "What number do you want to raise to the
power of 14?...", NUM
□A=NUM^14
□B=STR$(A)
□EX=SUBSTR("E",B)
□SIGN=MID$(B,EX+1,1)
□PLACES=RIGHT$(B,LEN(B)-EX-1)
□FOR T=1 TO LEN(B)
□C=MID$(B,T,1)
□IF C="." THEN
□DECIMAL=0
□GOTO 10
□ENDIF
□DECIMAL=DECIMAL+1
□IF C="E" THEN 100
□NEW=NEW+C
10□NEXT T
100□NEWCOUNT=VAL(PLACES)-DECIMAL
□NEW=NEW+LEFT$(ZERO,NEWCOUNT+1)
□FOR T=LEN(NEW) TO 1 STEP -1
□COUNT=COUNT+1
□NEWEST=MID$(NEW,T,1)+NEWEST
□IF MOD(COUNT,3)=2 AND T>1 THEN
□NEWEST=","+NEWEST
□ENDIF
□NEXT T
□PRINT NUM; " to the power of 14 = "; a
□PRINT "= "; NEWEST
□END
```

SUBSTR Searches for specified characters in a string

Syntax: **SUBSTR(*targetstring,searchstring*)**

Function: Searches for the first occurrence of *targetstring* within *searchstring* and returns the numeric value of its location. SUBSTR counts the first character in *searchstring* as character Number 1. Therefore, if you searched for the string "worth" in the string "Fortworth", SUBSTR returns a value of 5.

If SUBSTR cannot find *targetstring*, it returns a value of 0.

Parameters:

targetstring The group of characters you want to locate.

searchstring The string in which you want to find *targetstring*.

Examples:

```
PRINT SUBSTR("THREE","ONETWOTHREEFOURFIVESIX")

X=SUBSTR(" ",FULLNAME$)
```

Sample Program:

This procedure selects the last name from a string containing both a first name and a last name. It uses SUBSTR to find the space between the two names in order to determine where the last name begins.

```
PROCEDURE lastname
□DIM NAMES:STRING; LASTNAME:STRING[10]
□PRINT "Here are the last names:"
□FOR T=1 TO 10
□READ NAMES
□POINTER=SUBSTR(" ",NAMES)
□POINTER=LEN(NAMES)-POINTER
□LASTNAME=RIGHT$(NAMES,POINTER)
□PRINT LASTNAME
□NEXT T
□DATA "Joe Blonski","Mike Marvel","Hal
Skeemish","Fred Laungly"
□DATA "Jane Misty","Wendy Paston","Martha
Upshong","Jacqueline Rivers"
□DATA "Susy Reetmore","Wilson Creding"
□END
```

SYSCALL Executes an OS-9 System Call

Syntax: SYSCALL ***callcode registers***

Function: Lets you execute any OS-9 system call from BASIC09. Use this command to directly manipulate your system or data or to directly access devices.

Be careful! Used improperly, SYSCALL can cause undesirable results—you might unintentionally format a disk or destroy disk or memory data. Before using SYSCALL, you should be familiar with assembly language programming and should understand the system call information in the *OS-9 Technical Reference* manual. The *OS-9 Technical Reference* manual provides information about all OS-9 system calls.

To pass required register values to the SYSCALL command, create a complex data structure that contains values for all registers. For example:

```
TYPE REGISTERS=CC,A,B,DP:BYTE; X,Y,U:INTEGER
DIM REGS:REGISTERS
DIM CALLCODE:BYTE
```

The complex data type REGISTERS contains values for all registers. Unless you specifically assign values to variables (for instance, REGS.CC, REGS.A, and REGS.B in the previous example), they contain random values. See the TYPE command for further information.

Parameters:

callcode is the request code of the system call you wish to use. All system call codes are referenced in the *OS-9 Technical Reference* manual.

registers is a list of the register entry values required for the system call you are using.

Examples: see "Sample Programs."

Sample Programs:

The following programs set up a data type (REGISTERS) for the register variables. Before executing SYSCALL, the procedures store the required register entry values in the complex data structure REGS. The procedures also establish CALLCODE as a variable to hold the request code of the system call you want to use.

The Writecall procedure uses the string variable TEST to store text that it writes to the screen through Path 0 (the standard output path) using System Call $8A, I$Write. Before the procedure calls I$Write, it stores the appropriate path number (0) in Register A. The ADDR command calculates the address of the variable TEST, and the LEN command determines the length of the variable. The procedure stores these two values in Registers X and Y.

The Readcall uses System Call $8B, I$ReadLn to perform a function that is the opposite of Writecall. Readcall establishes TEST as a string variable into which it writes the characters you type. Because the length of TEST is restricted to ten characters (`DIM TEST:STRING[10]`), the terminal bell sounds if you attempt to type more than 10 characters. Pressing **[ENTER]** terminates the call and the procedure prints the contents of TEST—the characters you typed.

```
PROCEDURE WriteCall
□TYPE REGISTERS=CC,A,B,DP:BYTE; X,Y,U:INTEGER
□DIM REGS:REGISTERS
□DIM PATH,CALLCODE:BYTE
□DIM TEST:STRING[50]
□TEST="This is a test of I$Write, System call
$8A..."
□REGS.A=0
□REGS.X=ADDR(TEST)
□REGS.Y=LEN(TEST)
□CALLCODE=$8A
□RUN SYSCALL(CALLCODE,REGS)
□PRINT
□END

PROCEDURE Readcall
□TYPE REGISTERS=CC,A,B,DP:BYTE; X,Y,U:INTEGER
□DIM REGS:REGISTERS
□DIM PATH,CALLCODE:BYTE
□DIM TEST:STRING[10]
□REGS.A=0
□REGS.X=ADDR(TEST)
□REGS.Y=10
□CALLCODE=$8B
□RUN SYSCALL(CALLCODE,REGS)
□PRINT
□PRINT TEST
□END
```

TAB Causes PRINT to jump to the specified column

Syntax: TAB(*number*)

Function: Causes PRINT to display the next PRINT item to display in the column specified by *number*. If the cursor is already past the desired tab position, BASIC09 ignores TAB.

Use POS to determine the current cursor position when displaying characters on the screen.

Screen display columns are numbered from 1, the leftmost column, to a maximum of 255. The size of BASIC09 output buffer varies according to the stack size.

You can also use TAB with PRINT USING statements.

Parameters:

number	The column at which you want PRINT to begin.

Examples:

```
PRINT TAB(20);TITLE$
```

```
PRINT TAB(X);ITEMNUMBER;ITEM$
```

Sample Program:

This procedure uses asterisks to simulate a sine wave on the screen. It uses TAB to position each asterisk in the proper location.

```
PROCEDURE sinewave
□DIM FORMULA,CALCULATE,POSITION:REAL
□SHELL "DISPLAY 0C"
□FORMULA=(PI+2)/15
□CALCULATE=FORMULA
□SHELL "TMODE -PAUSE"
□FOR T=0 TO 200
□CALCULATE=CALCULATE+SQ(FORMULA)
□POSITION=INT(SIN(CALCULATE)*12+16)
□PRINT TAB(POSITION); "*"
□NEXT T
□SHELL "TMODE PAUSE"
□END
```

TAN Returns the tangent of a value

Syntax: TAN(***number***)

Function: Calculates the trigonometric tangent of *number*. Using DEG or RAD, you can specify the measure of the angle (*number*) in either degrees or radians. Radians are the default units.

Parameters:

number The angle for which you want to find the tangent.

Examples:

```
PRINT TAN(45)
```

Sample Program:

This procedure calculates sine, cosine, and tangent values for a number you type.

```
PROCEDURE ratiocalc
□DEG
□DIM ANGLE:REAL
□INPUT "Enter the angle of two sides of a
triangle...",ANGLE
□PRINT
□PRINT "Angle","SINE","COSINE","TAN"
□PRINT "-----------------------------------------
-------------------"
□PRINT ANGLE,SIN(ANGLE),COS(ANGLE),TAN(ANGLE)
□PRINT
□END
```

TRIM$ Removes spaces from the end of a string

Syntax: **TRIM$(*string*)**

Function: Removes any trailing spaces from the end of the specified string. This function is particularly useful for trimming records you recover from a random access file.

Parameters:

string	The string or string variable from which you wish to remove trailing spaces.

Examples:

```
PRINT TRIM$(A$)

GET A$,B$,C$
PRINT TRIM$(A$),TRIM$(B$),TRIM$(C$)
```

Sample Program:

This program takes names you type and puts them in a random access disk file. Because random access files use the same amount of storage for each item, short names are *padded* with extra spaces. When reading the names back from the file, the procedure uses TRIM$ to remove these extra spaces.

```
PROCEDURE namestor
□DIM NAMES,TEMP1,NAME(100):STRING[26]; FIRST,LAST:
STRING[15]; INITIAL:STRING[1]
□DIM PATH,T:INTEGER
□NAMES=""
□ON ERROR GOTO 10
□DELETE "namelist"
10□ON ERROR
□CREATE #PATH,"namelist":UPDATE
□FOR T=1 TO 100
□NAME(T)=""
□NEXT T
□T=0
```

```
□LOOP
□INPUT "First Name: ",FIRST
□EXITIF FIRST="" THEN
□CLOSE #PATH
□GOTO 100
□ENDEXIT
□INPUT "Initial: ",INITIAL
□INPUT "Last: ",LAST
□T=T+1
□NAME(T)=FIRST+" "+INITIAL+" "+LAST
□PUT #PATH,NAME(T)
□SEEK #PATH,T*26
□ENDLOOP
100□OPEN #PATH,"namelist":READ
□PRINT  \ PRINT
□PRINT "Lastname","Firstname","Initial"
□REM Print underline (40 characters)
□PRINT "________________________________________"
□PRINT
□SEEK #PATH,0
□T=0
□WHILE NOT(EOF(#PATH)) DO
□GET #PATH,NAMES
□T=T+1
□NAMES=TRIM$(NAMES)
□X=SUBSTR(" ",NAMES)
□FIRST=LEFT$(NAMES,X-1)
□TEMP1=RIGHT$(NAMES,LEN(NAMES)-X+1)
□INITIAL=MID$(TEMP1,2,1)
□LAST=RIGHT$(TEMP1,LEN(TEMP1)-3)
□PRINT LAST,FIRST,INITIAL
□SEEK #PATH,T*26
□ENDWHILE
□CLOSE #PATH
□END
```

TRON/TROFF **Turns on/off trace function**

Syntax: **TRON**
TROFF

Function: Turns on or off the BASIC09 trace mode. When trace is turned on (TRON), BASIC09 decompiles and displays each statement in a procedure before execution. BASIC09 also displays the result of each expression after evaluation. This function lets you follow program flow and is helpful in debugging and analyzing the execution of a procedure. After the procedure is debugged, remove the TRON statement.

If you want to view only a portion of a procedure's execution, surround that portion with TRON and TROFF. Tracing begins immediately after the TRON statement and ends at the TROFF statement. The rest of the program executes normally.

Parameters: None

Examples:

```
B$="0000000000000000000000"+B$
N=1+LEN(B$)
FOR I=4 TO 1 STEP -1
TRON
N=N-4
A(I)=VAL(MID$(B$,N,4))
TROFF
NEXT I
```

TRUE **Returns a Boolean TRUE value**

Syntax: ***variable*=TRUE**

Function: TRUE is a Boolean function that always returns True. You can use TRUE and FALSE to assign values to Boolean variables.

Parameters:

variable	The Boolean storage unit you want to set to True.

Examples:

```
DIM TEST:BOOLEAN
TEST=TRUE
```

Sample Program:

This procedure asks five questions. If your answer is correct, it stores the Boolean value TRUE in a Boolean type variable. If your answer is incorrect, it stores the Boolean value FALSE in the variable.

```
PROCEDURE quiz
□DIM REPLY,VALUE:BOOLEAN; ANSWER:STRING[1];
QUESTION:STRING[80]
□FOR T=1 TO 5
□READ QUESTION,VALUE
□PRINT QUESTION
□PRINT "(T) = TRUE□□□□□□□□(F) = FALSE"
□PRINT "Select T or F:□□";
□GET #1,ANSWER
□IF ANSWER="T" THEN
□REPLY=TRUE
□ELSE
□REPLY=FALSE
□ENDIF
□IF REPLY=VALUE THEN
□PRINT \ PRINT "That's Correct...Good Show!"
□ELSE
```

```
□PRINT "Sorry, you're wrong...Better Luck next
time."
□ENDIF
□PRINT  \ PRINT  \ PRINT
□NEXT T
□DATA "In computer talk, CPU stands for Central
Packaging Unit.", FALSE
□DATA "The actual value of 64K is 65536
bytes.",TRUE
□DATA "The bits in a byte are normally numbered Ø
through 7.",TRUE
□DATA "BASICØ9 has four data types.",FALSE
□DATA "The LAND function is a Boolean type
operator.",FALSE
□END
```

TYPE **Defines a data type**

Syntax: TYPE ***name* = *typedeclar*** [;***typedeclar***[;...]]

Function: Defines new data types (complex data structures). New data types are *vectors* (one-dimensional arrays) of previously defined types. Structures created by TYPE differ from arrays in that they can consist of elements of different types, and BASIC09 accesses elements by field names rather than by an indexed position.

Parameters:

name	The name you select for the new data type.
typedeclar	One or more type declarations, which can consist of field names, type declarations, and subscripts. Separate different types or different lengths of string declarations with semicolons.

Notes:

- Complex data structures allow you to create data types that are appropriate for a specific task. You can organize, read, and write associated data naturally. Also, BASIC09 establishes and defines element positions at compilation time. This saves time and overhead at run time because BASIC09 can access the elements of a data structure faster than it can access the elements of an array.
- When you define new data structures using TYPE, you can include any of the five existing data types (string, real, integer, byte, and Boolean), or you can include data structure types that you previously defined with TYPE. This means that your structures can be simple or very complex, such as non-rectangular data lists or trees.
- TYPE does not create storage. You create storage using the DIM statement, after using TYPE.
- To access elements of a data structure, use the field name as well as any appropriate element index.

- For more information on creating and using complex data types, see "Complex Data Types" in Chapter 6.

Examples:

```
TYPE LIBRARY=TITLE,AUTHOR,PUBLISHER:STRING[25];
REFERENCE:INTEGER
DIM BOOK(500):LIBRARY

TYPE PARTS=ITEM,LOCATION:STRING[20]; CAT:REAL;
QUANTITY;INTEGER
DIM INVENTORY(1000):PARTS
```

Sample Program:

This procedure builds an array to contain a book reference list, including the book title, the author's name, the publisher, and a reference number. It does so by using TYPE to create a special data structure to store all the information for each book.

```
PROCEDURE books
□TYPE LIBRARY=TITLE,AUTHOR,PUBLISHER:STRING[30];
REFERENCE:INTEGER
□DIM BOOKS(100):LIBRARY
□T=0
□LOOP
□T=T+1
□INPUT "BOOK TITLE...",BT$
□BOOKS(T).TITLE=BT$
□EXITIF BOOKS(T).TITLE="" THEN
□GOTO 100
□ENDEXIT
□INPUT "Book Author...",BA$
□BOOKS(T).AUTHOR=BA$
□INPUT "Book Publisher...",BP$
□BOOKS(T).PUBLISHER=BP$
□INPUT "Reference Number...",BOOKS(T).REFERENCE
□ENDLOOP
100□FOR X=1 TO T-1
□PRINT BOOKS(X).TITLE; " , "; BOOKS(X).AUTHOR; " ,
";
□BOOKS(X).PUBLISHER; " , "; BOOKS(X).REFERENCE
□NEXT X
□END
```

UNTIL Terminates a REPEAT loop on specified condition

Syntax: **REPEAT**
procedure lines
UNTIL *expression*

Function: Ends a REPEAT loop. REPEAT establishes a loop that executes the encompassed procedure lines until the result of the expression following UNTIL is true. Because the loop is tested at the bottom, the lines within the loop are executed at least once.

Parameters:

procedures lines	Statements you want to execute in the loop.
expression	A Boolean expression (the result must be either True or False).

Examples:

```
REPEAT
COUNT = COUNT+1
UNTIL COUNT > 100

INPUT X,Y
REPEAT
X = X-1
Y = Y-1
UNTIL X<1 OR Y<0
```

See REPEAT for more information.

USING **Formats PRINT output**

Syntax: **PRINT [#*path*] USING [*format*,] *data*[;*data*...]**

Function: Prints data using a format you specify. This statement is especially useful for printing report headings, accounting reports, checks, or any document requiring a specific format.

USING is actually an extension of the PRINT statement. The same rules that apply to the PRINT statement also apply to the PRINT USING statement (see PRINT).

Parameters:

path	The number to an opened device or file. If you do not specify *path* the default is #1, the video screen (standard output device). To print to another device or file, first OPEN a path to that file or device (see OPEN).
format	An expression specifying the arrangement of the displayed data.
data	Any numeric or string constant or variable. Always enclose string constants within quotation marks. Separate all data items with semicolons or commas.

See PRINT USING for more information.

VAL **Converts string data to numeric data**

Syntax: VAL(***string***)

Function: Converts string-type data to numeric-type. VAL is the inverse of the STR$ function. It returns the real value represented by the characters in a string. If any character in the specified string is not numeric, BASIC09 returns an error.

Parameters:

string	An ASCII string containing one or more of the following characters: 0123456789.+$-.

Examples:

```
PRINT VAL(123)

A$="44.66"
PRINT VAL(A$)
```

Sample Program:

This procedure calculates an exponential value, then adds the necessary number of zeroes to convert it to standard notation. It uses STR$ to convert the original number to a string, then uses VAL to convert the exponent into a value to determine the correct decimal place.

```
PROCEDURE bignum
□DIM  C,PLACES,B,SIGN:STRING;  EX,COUNT,NEWCOUNT,
DECIMAL:INTEGER
□DIM NEW,ZERO,NEWEST:STRING[100]
□COUNT=-1
□ZERO="000000000000000000000000000000000000"
□NEW="" \NEWEST=""
□INPUT "What number do you want to raise to the
power of 14?...",NUM
□A=NUM^14
□B=STR$(A)
□EX=SUBSTR("E",B)
□SIGN=MID$(B,EX+1,1)
```

```
□PLACES=RIGHT$(B,LEN(B)-EX-1)
□FOR T=1 TO LEN(B)
□C=MID$(B,T,1)
□IF C="." THEN
□DECIMAL=0
□GOTO 10
□ENDIF
□DECIMAL=DECIMAL+1
□IF C="E" THEN 100
□NEW=NEW+C
10□NEXT T
100□NEWCOUNT=VAL(PLACES)-DECIMAL
□NEW=NEW+LEFT$(ZERO,NEWCOUNT+1)
□FOR T=LEN(NEW) TO 1 STEP -1
□COUNT=COUNT+1
□NEWEST=MID$(NEW,T,1)+NEWEST
□IF MOD(COUNT,3)=2 AND T>1 THEN
□NEWEST=","+NEWEST
□ENDIF
□NEXT T
□PRINT NUM; " to the power of 14 = "; A
□PRINT "= "; NEWEST
□END
```

WHILE/DO/ENDWHILE **Establishes a loop**

Syntax: **WHILE *expression* DO**
procedure lines
ENDWHILE

Function: Establishes a loop that executes the encompassed procedure lines while the result of the expression following WHILE is true. Because the loop is tested at the top, the lines within the loop are never executed unless the expression is true.

Parameters:

expression	A Boolean expression (has a result of True or False).
procedure lines	Program lines to execute if the expression is true.

Examples:

```
WHILE COUNT < 12 DO
COUNT = COUNT+1
ENDWHILE
```

Sample Program:

You must create a file of directory names using the GET sample program before you can use the following procedure. Copyutil uses the filenames created by the GET sample program to copy a directory's files to another directory you specify. You must specify a directory name that does not exist. Copyutil uses a WHILE/DO/ENDWHILE loop to continue copying until BASIC09 reaches the end of the file.

```
PROCEDURE copyutil
□DIM PATH,T,COUNT:INTEGER; FILE,JOB,DIRNAME:STRING
□OPEN #PATH,"dirfile":READ
□INPUT "Name of new directory...",DIRNAME
□SHELL "MAKDIR "+DIRNAME
□SHELL "LOAD COPY"
□WHILE NOT(EOF(#PATH)) DO
□READ #PATH,FILE
□JOB=FILE+" "+DIRNAME+"/"+FILE
□ON ERROR GOTO 10
□PRINT "COPY "; JOB
□SHELL "COPY "+JOB
10□ON ERROR
□ENDWHILE
□CLOSE #PATH
□END
```

WRITE **Writes data to a sequential file or device**

Syntax: **WRITE [*#path,*] *data***

Function: Writes an ASCII record to a sequential file or to a device.

Parameters:

path	A variable containing the path number of the file or device to which you want to send data. *Path* can be one of the the standard I/O paths (0, 1, 2).
data	The data you want to send over the specified path.

Notes:

The following information deals with writing sequential disk files:

- To write file records, you must first dimension a variable to contain the path number of the file, then use OPEN or CREATE to open a file in the WRITE or UPDATE access mode.
- Records can be of any length within a file.
- Individual data items in the input record are separated by ASCII null characters. You can also separate numeric items with comma or space character delimiters. Each input record is terminated by a carriage return character.

Examples:

```
WRITE #PATH,DATA$

WRITE #1,RESPONSE$

WRITE #OUTPUT,INDEX(X)
```

```
OPEN #PATH,"namefile":WRITE
FOR T=1 TO 10
READ NAME$
WRITE #PATH, NAME$
NEXT T
CLOSE #PATH
DATA "JIM","JOE","SUE","TINA","WENDY"
DATA "SALL","MICKIE","FRED","MARV","WINNIE"
```

Sample Program:

This procedure selects 100 random values between 1 and 10. It uses WRITE to place the values into a disk file. Next, it reads the values from the file and uses asterisks to indicate how many times RND selected each value.

```
PROCEDURE randlist
□DIM SHOW,BUCKET:STRING
□DIM T,PATH,SELECT(10),R:INTEGER
□BUCKET="************************"
□FOR T=1 TO 10
□SELECT(T)=0
□NEXT T
□ON ERROR GOTO 10
□SHELL "DEL RANDFILE"
10□ ON ERROR
□CREATE #PATH,"randfile":UPDATE
□FOR T=1 TO 100
□R=RND(9)+1
□WRITE #PATH,R
□NEXT T
□PRINT "Random Distribution"
□SEEK #PATH,0
□FOR T=1 TO 100
□READ #PATH,NUM
□SELECT(NUM)=SELECT(NUM)+1
□NEXT T
□FOR T=1 TO 10
□SHOW=RIGHT$(BUCKET,SELECT(T))
□PRINT USING "S6<,I3<,S2<,S20<","Number",T,":",
SHOW
□NEXT T
□CLOSE #PATH
□END
```

XOR Returns the exclusive OR of two values

Syntax: ***operand1* XOR *operand2***

Function: Performs the logical exclusive OR operation on two or more values, returning a value of either TRUE or FALSE.

Parameters:

operand1 *operand2*	Boolean values or expressions (that result in values of True or False).

Examples:

```
PRINT A>2 XOR B>3

PRINT A$="YES" XOR B$="YES"
```

Sample Program:

This procedure lets two people type numbers until one of them guesses the number that the computer picks. It uses XOR to determine that one of the numbers typed is the correct number, but not both.

```
PROCEDURE drawstraw
□DIM NUM1,NUM2,R:INTEGER; A:BOOLEAN
□PRINT "This program will help you pick"
□PRINT "between two people. Choose who will be"
□PRINT "Person 1 and who will be Person 2."
□PRINT "Then, enter numbers between 1 and 10"
□PRINT "when requested."
□PRINT
□R=RND(10)
10□INPUT "Person 1, type a number and press
ENTER...",NUM1
□INPUT "Person 2, type a number and press
ENTER...",NUM2
□A=NUM1=R XOR NUM2=R
□IF A=FALSE THEN
□PRINT "You'll have to try again..."
□PRINT
```

```
□GOTO 10
□ENDIF
□IF NUM1=R THEN
□PRINT "You win, Person 1"
□ENDIF
□IF NUM2=R THEN
□PRINT "You win, Person 2"
□ENDIF
□PRINT "The Number was..."; R
□END
```

Chapter 12

Program Optimization

BASIC09's multipass compiler produces a compressed and optimized low-level I-code for execution. Compared to other BASIC languages, BASIC09 greatly decreases both the storage space required for program code and the execution speed of programs.

Because BASIC09 produces I-code at a powerful level, it can handle numerous MPU (micro processor unit) instructions with a single interpretation. Therefore, for complex programs, there is little performance difference between the execution of I-code and pure machine-language instructions.

Most BASIC languages have to compile from text as they run, or search tables of *tokens* in order to execute code. Instead, BASIC09 I-code instructions contain direct references to variables, statements, and labels. BASIC09 fully utilizes the power of the 6809 instruction set, as well, which is optimized for efficient execution of compiler-produced code.

Because BASIC09 interprets I-code, you have a variety of entry-time and run-time tests and development aids. The editor reports syntax errors immediately when they are entered. The debugger lets you debug using original program source statements and names. The I-code interpreter performs run-time error checking of array structures and BASIC09 functions.

Optimum Use of Numeric Data Types

The following notes apply to the use of BASIC09 numeric data types:

- BASIC09 includes several different numeric representations (real, integer, byte), and performs automatic type conversions between them. This means that without care, your code might contain expressions or loops that take more than ten times longer to execute than is necessary.

- Some BASIC09 numeric operators, such as +,-,*, and /, and some BASIC09 control structures include versions for both real and integer values. Integer versions execute much faster and can have slightly different properties. For instance, integer division discards any remainder.

  Integer operations are faster because they use corresponding 6809 instructions. Using integers increases speed and decreases storage requirements. Integer operations use the same symbols as real operations, but BASIC09 automatically selects the integer operations when when all operands of an expression are of byte or integer type.

- Type conversion takes time. Using expressions with operands and operators of the same kind is most efficient.

- BASIC09's real (floating point) math provides excellent performance. It includes a 40-bit binary floating point representation and uses the CORDIC technique to derive all transcendental functions. This integer shift-and-add technique is faster and more consistent than the common series-expansion approximations.

- At times, you can obtain similar or identical results in a number of different ways and at different execution speeds. For example, if the variable Value is an integer, then `Value*2` is a fast integer operation. However, if the expression is `Value*2.0`, 2.0 is represented as a real number and the operation requires real multiplication. BASIC09 must transform the integer Value into a real value. If the result of the expression is assigned to an integer type variable, BASIC09 must transform the result back to an integer type. The decimal point can slow the operation by about ten times.

Arithmetic Functions Ranked by Speed

Operation	Typical Speed in MPU Cycles
Integer add or subtract	150
Integer multiply	240
Real add	440
Real subtract	540
Integer divide	960
Real multiply	990
Real divide	3870
Real square root	7360
Real logarithm or exponential	20400
Real sine or cosine	32500
Real power	39200

Referring to the previous table can help you in your programming. For instance, notice that it is quicker to add a value to itself rather than multiplying it by 2. Similarly, multiplying a value by itself or using SQ on a value is much faster than raising a value to the power of 2.

Notice that a real divide takes 3870 cycles, while a real multiplication takes only 990 cycles. Multiplying a value by 0.5 is four times quicker than dividing the value by 2.

Quicker Loops

BASIC09 has two versions of FOR/NEXT loops, one for integer loop counter variables and one for real loop counter variables. It automatically uses the appropriate version. Integer FOR/NEXT loops are much faster than real FOR/NEXT loops.

Other kinds of loops also run faster if you use integer type variables for the loop counters. When writing program loops, remember that statements inside the loop can execute many times for each execution outside the loop. Whenever possible, compute values before entering loops.

Arrays and Data Structures

The internal workings of BASIC09 use integer numbers to index arrays and complex data structures. This means that BASIC09 must convert real type variable or expression subscripts before it can handle them. Using integer expressions for subscripts increases execution speed.

Using the assignment statement LET to copy identically sized data structures is much faster than copying arrays or structures element-by-element inside a loop.

The PACK Command

PACK causes a second compilation of a specified procedure. Depending on such variables as the number of procedure comments and the inclusion of line numbers, packed procedures execute from 10 to 30 percent faster. Line numbers cause unpacked procedures to run slower.

Minimizing Constant Expressions and Subexpressions

For maximum execution speed, precalculate constant expressions. For instance, `x = x+5` produces the same result as `x = x+sqrt(100)/2`. However, the first expression requires approximately 150 MPU cycles while the second expression requires 11,650 MPU cycles. If you use such an expression inside a loop, the additional execution time is enormous.

Input and Output

Accessing data one line or record at a time is much faster than accessing it one character at a time. Also, the GET and PUT statements are much faster than READ and WRITE statements when accessing disk files. This is because GET and PUT use the same binary format as BASIC09's internal operations. READ, WRITE, PRINT, and INPUT must perform binary-to-ASCII or ASCII-to-binary conversions, which take more time.

Error Codes

Signal Errors

Code	Meaning
1	Unconditional termination
2	Keyboard termination
3	Keyboard interrupt

BASIC09 Error Codes

Code	**Meaning**
10	Unrecognized symbol
11	Excessive verbiage
12	Illegal statement construction
13	I-code overflow, need more workspace memory
14	Illegal channel reference, bad path number given
15	Illegal mode (read/write/update) - directory only
16	Illegal number
17	Illegal prefix
18	Illegal operand
19	Illegal operator
20	Illegal record field name
21	Illegal dimension
22	Illegal literal
23	Illegal relational
24	Illegal type suffix
25	Too-large dimension
26	Too-large line number
27	Missing assignment statement
28	Missing path number
29	Missing comma
30	Missing dimension
31	Missing DO statement
32	Memory full, need more workspace memory
33	Missing GOTO
34	Missing left parenthesis
35	Missing line reference
36	Missing operand
37	Missing right parenthesis
38	Missing THEN statement
39	Missing TO

Code	Meaning
40	Missing variable reference
41	No ending quote
42	Too many subscripts
43	Unknown procedure
44	Multiply-defined procedure
45	Divide by zero
46	Operand type mismatch
47	String stack overflow
48	Unimplemented routine
49	Undefined variable
50	Floating overflow
51	Line with compiler error
52	Value out of range for destination
53	Subroutine stack overflow
54	Subroutine stack underflow
55	Subscript out of range
56	Parameter error
57	System stack overflow
58	I/O type mismatch
59	I/O numeric input format bad
60	I/O conversion: number out of range
61	Illegal input format
62	I/O format repeat error
63	I/O format syntax error
64	Illegal path number
65	Wrong number of subscripts
66	Non-record-type operand
67	Illegal argument
68	Illegal control structure
69	Unmatched control structure
70	Illegal FOR variable
71	Illegal expression type
72	Illegal declarative statement
73	Array size overflow
74	Undefined line number
75	Multiply-defined line number
76	Multiply-defined variable
77	Illegal input variable
78	Seek out of range
79	Missing data statement

Windowing and System Errors

Code	Meaning
183	Illegal window type
184	Window already defined
185	Font not found
186	Stack overflow
187	Illegal argument
188	(unused)
189	Illegal coordinates
190	Internal integrity check
191	Buffer size is too small
192	Illegal command
193	Screen or window table is full
194	Bad/undefined buffer number
195	Illegal window definition
196	Window undefined
197	(unused)
198	(unused)
199	(unused)
200	Path table full
201	Illegal path number
202	Interrupt polling table full
203	Illegal mode
204	Device table full
205	Illegal module header
206	Module directory full
207	Memory full
208	Illegal service request
209	Module busy
210	Boundary error
211	End of file
212	Returning non-allocated memory
213	Non-existing segment
214	No permission
215	Bad path name
216	Path name not found
217	Segment list full
218	File already exists
219	Illegal block address
220	Phone hangup data carrier detect lost
221	Module not found
223	Suicide attempt

Code	Meaning
224	Illegal process number
226	No children, can't wait for nonexistent child process
227	Illegal SWI code
228	Process aborted, signal 2
229	Process table full, can't fork a process
230	Illegal parameter area
231	Known module
232	Incorrect module CRC
233	Signal error
234	Non-existent module
235	Bad name
237	System RAM full
238	Unknown process ID
239	No task number available
240	Illegal unit error
241	Bad sector number
242	Write protected disk
243	CRC error
244	Read error
245	Write error
246	Not ready, device not ready
247	Seek error
248	Media full
249	Wrong type, incompatible media type
250	Device busy
251	Disk ID change, disk changed with open files
252	Record is locked out
253	Non-sharable file busy

The Inkey Program

Assembly Language Listing of Inkey

An assembled version of Inkey is included on the CONFIG/ BASIC09 diskette. Use Inkey from BASIC09 with the RUN statement.

```
***********************
* INKEY - a subroutine for BASIC09 by Robert Doggett
*
* Called by: RUN INKEY(StrVar)
*            RUN INKEY(Path,StrVar)
* INKEY determines if a key has been typed on the given path
* (Standard Input if not specified), and if so, returns the next
* character in the String Variable. If no key has been typed, the
* null string is returned. If a path is specified, it must be
* either type BYTE or INTEGER.

                                 NAM     INKEY
                                 IFP1
                                 USE     /D0/DEFS/OS9DEFS
                                 ENDC

  0021               TYPE     set     SBRTN+OBJCT
  0081               REVS     set     REENT+1

  0000  87CD005E              mod     InKeyEnd,InKeyNam,TYPE,REVS
                                      ,InKeyEnt,SIZE
  000D  496E6B65     InKeyNam fcs     "Inkey"
D 0000                        org     0          Parameters
D 0000               Return   rmb     2          Return addr of caller
D 0002               PCount   rmb     2          Num of params following
D 0004               Param1   rmb     2          1st param addr
D 0006               Length1  rmb     2          size
D 0008               Param2   rmb     2          2nd param addr
D 000A               Length2  rmb     2          size
  000C               E$Param  equ     $38
  000E               SIZE     equ     *
  0012  3064         InKeyEnt leax    Param1,S
```

```
0014  EC62                ldd    Pcount,S     Get parameter count
0016  10830001            cmpd   #1           just one parameter?
001A  2727                beq    InKey20      ..Yes; default path A=0
001C  10830002            cmpd   #2           Are there two params?
0020  2635                bne    ParamErr     No, abort
0022  ECF804              ldd    [Param1,S]   Get path number
0025  AE66                ldx    Length1,S
0027  301F                leax   -1,X         byte variable?
0029  2706                beq    InKey10      ..Yes; (A)=Path number
002B  301F                leax   -1,X         Integer?
002D  2628                bne    ParamErr     ..No; abort
002F  1F98                tfr    B,A
0031  3068      InKey10   leax   Param2,S
0033  EE02      InKey20   ldu    2,X          length of string
0035  AE84                ldx    0,X          addr of string
0037  C6FF                ldb    #$FF
0039  E784                stb    0,X          Initialize to null str
003B  11830002            cmpu   #2           at least two-byte str?
003F  2502                blo    InKey30      ..No
0041  E701                stb    1,X          put str terminator
0043  C601      InKey30   ldb    #SS.Ready
0045  103F8D              OS9    I$GetStt     is there any data ready?
0048  2508                bcs    InKey90      ..No; exit
004A  108E0001            ldy    #1
004E  103F89              OS9    I$Read       Read one byte
0051  39                  rts                 returns error status
0052  C1F6      InKey90   cmpb   #E$NotRdy
0054  2603                bne    InKeyErr
0056  39                  rts                 (carry clear)
0057  C638      ParamErr  ldb    #E$Param     Parameter Error
0059  43        InKeyErr  coma
005A  39                  rts
005B  1A6916              emod
005E            InKeyEnd  equ    *
```

Index